Semantic-Enhanced Static Vulnerability Detection in Baseband Firmware

Yiming Liu[*,†]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
liuyiming@iie.ac.cn

Cen Zhang
Nanyang Technological University
Singapore
cen001@e.ntu.edu.sg

Feng Li[*,†,‡]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
lifeng@iie.ac.cn

Yeting Li[*,†]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
liyeting@iie.ac.cn

Jianhua Zhou[*,†]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
zhoujianhua@iie.ac.cn

Jian Wang[*,†]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
wangjian411x@iie.ac.cn

Lanlan Zhan[*,†]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
zhanlanlan@iie.ac.cn

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

Wei Huo[*,†]
Institute of Information Engineering,
CAS; School of Cyber Security, UCAS
Beijing, China
huowei@iie.ac.cn

ABSTRACT

Cellular network is the infrastructure of mobile communication. Baseband firmware, which carries the implementation of cellular network, has critical security impact on its vulnerabilities. To handle the inherent complexity in cellular communication, cellular protocols are usually implemented as message-centric systems, containing the common message processing phase and message specific handling phase. Though the latter takes most of the code (99.67%) and exposed vulnerabilities (74%), it is rather under-studied: existing detectors either cannot sufficiently analyze it or focused on analyzing the former phase.

To fill this gap, we proposed a novel semantic-enhanced static vulnerability detector named BVFinder focusing on message specific phase vulnerability detection. Generally, it identifies a vulnerability by locating whether a predefined sensitive memory operation is tainted by any attacker-controllable input. Specifically, to reach high automation and preciseness, it made two key improvements: a semantic-based taint source identification and an enhanced taint propagation. The former employs semantic search techniques to identify registers and memory offsets that carry attacker-controllable inputs. This is achieved by matching the inputs to their corresponding message and data types using textual features and addressing patterns within the assemblies. On the other hand, the latter technology guarantees effective taint propagation by employing additional indirect call resolution algorithms.

The evaluation shows that BVFinder outperforms the state-of-the-art detectors by detecting three to four times of amount of vulnerabilities in the dataset. Till now, BVFinder has found four zero-day vulnerabilities, with four CVEs and 12,410 USD bounty assigned. These vulnerabilities can potentially cause remote code execution to phones using Samsung shannon baseband, affecting hundreds of millions of end devices.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security; Embedded systems security**.

KEYWORDS

Cellular Baseband, Static Taint Analysis, Vulnerabilities

ACM Reference Format:
Yiming Liu, Cen Zhang, Feng Li, Yeting Li, Jianhua Zhou, Jian Wang, Lanlan Zhan, Yang Liu, and Wei Huo. 2024. Semantic-Enhanced Static Vulnerability Detection in Baseband Firmware. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639158

1 INTRODUCTION

Wireless networks have developed rapidly in the past decade. According to statistics, more than 10 billion cellular devices have been connected to cellular networks [17]. These devices rely on baseband firmware, which runs on a baseband processor (BP) and handles radio communication and cellular protocol processing. The presence of vulnerabilities in this firmware from major vendors could

[*] Also with Key Laboratory of Network Assessment Technology, CAS.
[†] Also with Beijing Key Laboratory of Network Security and Protection Technology.
[‡] Corresponding author.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639158

Table 1: Statistics of Two Message Processing Phases. VUL NUM **is analyzed from all exposed baseband vulnerabilities, and BB NUM shows the basic block data of Samsung G960FXXS2BRH8 firmware.**

Phase	VUL NUM	Percent	BB NUM	Percent
Common	33	26%	263	0.33%
Message-Specific	95	**74%**	80374	**99.67%**

pose serious security risks. For example, Google Project Zero [51] recently disclosured a multitude of critical vulnerabilities in Exynos modems, which could allow remote code execution without any user interaction. These issues have affected dozens of device models from various manufacturers, such as Google, Samsung, and Vivo, affecting hundreds of millions of end users.

To handle the inherent complexity of the communication, cellular protocols are divided into smaller sub-protocols and they communicate through messages, leading to a design that is predominantly message-centric. In this system, message processing consists of two phases: common message preprocessing (referred to as the common phase) and message-specific handling (referred to as the message-specific phase). The former phase manages the message queues and parses the basics of incoming messages, while the latter operates message-specific semantics. As shown in Table 1, for the key control panel L3 (layer 3) of cellular protocol implementation on the baseband firmware, the message-specific phase accounts for most of both the code implementation and the found vulnerabilities, covering hundreds of different types of messages.

Despite the significance of the message-specific phase, applying existing vulnerability detectors can acquire limited outcomes. Most detectors are dynamic fuzzing tools, either performing blackbox fuzzing towards physical devices [8, 22, 24, 30, 39] or utilizing advanced emulation techniques for scalable greybox fuzzing [20, 31, 33]. However, they cannot adequately test most message-specific logic since the effective testing for one type of message requires specific setups to reach the exact execution status. Preparing such setups involves not only executing prefix message sequences, but also configuring internal structures, which demand extensive manual efforts. Static analysis based detectors, which do not require intricate testing setups, are rather under-explored. The only existing static detectors, BaseSpec [29] and BaseComp [28] focus on detecting common phase vulnerabilities. The former proposed a semi-automatic approach to find the inconsistencies between the protocol specifications and common phase implementations while the latter is for analyzing integrity protection. Conclusively, building more effective message-specific vulnerability detectors [1] is a practical need.

To fill this gap, we first conducted an empirical study to understand the characteristics of message-specific vulnerabilities. The study includes all 225 exposed vulnerabilities from top three vendors with a time span of nine years. It reveals that most vulnerabilities: ❶ are memory safety issues caused by improper handling of message content; ❷ fit a straightforward detection pattern: attacker-controllable input + sensitive memory operations; ❸ have strong locality that only analyzing message-specific phase is sufficient for detection. These observations motivate us to design a static taint analysis based detector, which starts analysis from message-specific phase, marks the binary variables [2] carrying the attacker-controllable content as taint sources, and sets the assemblies matching sensitive memory operations as sink points.

Though the intuition behind is straightforward, it is quite challenging to guarantee its practicality in such a complex and closed-source baseband firmware. Overall, the high automation and effectiveness of the taint analysis need to be ensured in a limited binary analysis environment. On one hand, since only certain fields of particular types of messages are attacker-controllable, determining the taint sources needs the precise identification of whether a binary variable carries specific message content. This requires a semantic match in the involved binary variables' usages. On the other hand, the taint propagation usually terminated due to the unsolved indirect calls inside the control flow graph (abbr as CFG). This kind of early termination is particularly prevalent in baseband firmware due to its inherent complexity and message-centric design.

To solve these challenges, we proposed a semantic-enhanced taint analysis approach for static vulnerability detection. For automatically pinpointing the precise taint sources, the goal is to identify the binary variables whose data is of specific message and field type, *i.e.*, is attacker-controllable. Our main observation is that the assemblies which use this variable expose enough information for the identification. Specifically, for a binary variable, it is a qualified taint source if it passes two kinds of matches: the high level message type match, and the low level data type match. The former match is done by comparing the message description documented in 3GPP specification with the textual features contained in these assemblies, such as the log strings, function names, etc. The latter match is accomplished by extracting the addressing patterns from these assemblies and validating them with the expected addressing behaviours that accessing a target field in a parsed message should have. Overall, we first collected all binary variables outputted from the common phase, forming a superset of candidate taint sources. Then the sources which pass these two matches are marked as the final taint sources. For solving indirect calls, our basic observation is that the sources of most callees are function addresses hardcoded in binaries. Therefore, we proposed an algorithm to identify these callee sources, extract the caller-callee mappings between them and these unsolved indirect calls via inter-procedural data flow analysis, and finally use these mappings for effective taint propagation. Lastly, two types of sensitive operations are empirically summarized based on the studied vulnerabilities and the assemblies matching the operations are set as taint sink points.

We implemented a prototype tool named BVFinder and did further evaluation. In the dataset containing all existing vulnerabilities, BVFinder can detect all existing vulnerabilities, which is three to four times of the detection number compared with state-of-the-art detectors, showing its superior detection ability. The ablation study further demonstrate that each key component of BVFinder contributes significantly to the overall performance, where the semantic taint source identification raises the true positive rate and the enhanced taint propagation contributes in lowering the false negative rate. Finally, BVFinder is applied in latest baseband

[1] In this paper, we use message-specific vulnerability to represent the vulnerabilities whose root causes are in message-specific phase.

[2] We define registers, stack offsets or global data offsets as binary variables. The output binary variables of a function are those binary variables written by the function.

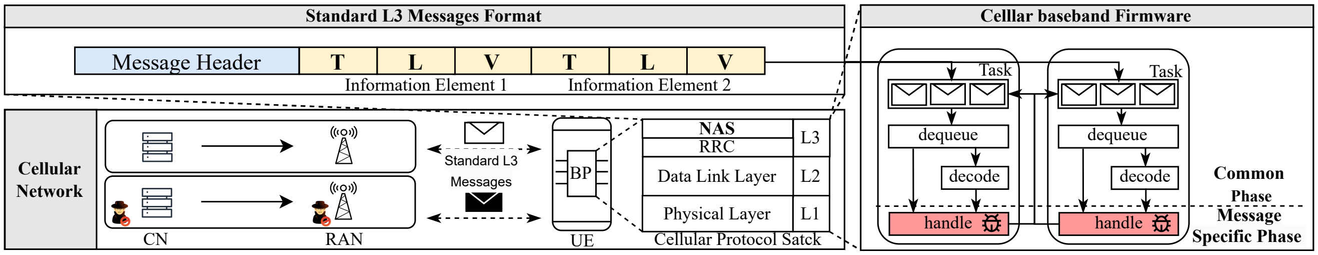


Figure 1: Overview of Cellular Protocol Based Communication

firmware of Samsung and Mediatek. Till now, it has found four new vulnerabilities (four CVEs and 12,410 USD bounty) where three of them can cause potential remote code execution, affecting hundreds of millions of end devices.

Our contributions are summarized as following:

- we conducted the first empirical study on understanding message specific vulnerabilities, highlighting its current vulnerability type distribution and detection pattern;
- we proposed and implemented a semantic-enhanced static vulnerability detector, which contains two key technical improvements on both taint source identification and taint propagation;
- we evaluated our detector with state-of-the-art tools and conducted detailed ablation study to understand the effectiveness of each component;
- we applied BVFinder to find four zero-day vulnerabilities which have critically high security impacts, affecting hundreds of millions of end users. We responsibly reported them to vendors and helped the fix.

To facilitate future research, we have released our studied dataset and the prototype [4].

2 PRELIMINARIES

2.1 Background

Cellular Protocols As shown in Figure 1, the cellular network is typically composed of core network (CN), radio access network (RAN), and user equipment (UE). These components communicate with each other via different cellular protocols depending on the cellular network generation, *e.g.*, 2G, 3G, 4G, or 5G. A user equipment refers to the end-user device or the mobile device used by individuals to access mobile communication networks such as smartphones. It employs the baseband processor (BP) to process radio signals and handles cellular protocol messages received from the radio access network. The primary role of a radio access network is to establish and maintain connectivity between user equipment and the core network of a cellular service provider. The core network is responsible for core processes including session management, mobility management, etc.

These components communicate on a 3-layer cellular protocol stack. Similar to the OSI model, the layer 1 is the physical layer and the layer 2 is the data link layer. Layer 3 includes the set of protocols used for the control and management of mobile network functions. The cellular protocol specification is defined and maintained by the 3GPP organization and it includes hundreds of specification documents. Among all cellular messages, standard L3 layer messages are used for key interactions between cellular components such as connection management (CM), session management (MM), authentication, etc. Information element (IE) is the basic unit of the standard L3 message. As shown in Figure 1, it consists of three fields: type (T), length (L), and value (V).

Table 2: Overview of Studied Vulnerabilities. **X(Y) represents the number of total(memory safety) vulnerabilities.**

Type	MediaTek	Samsung	Qualcomm	SUM	Percentage
Common Phase	1(1)	3(3)	29(25)	33(29)	14.7%
Message-Specific	4(3)	12(5)	79(54)	95(62)	42.2%
Unknown	8(8)	11(11)	78(58)	97(77)	43.1%
SUM	13(12)	26(19)	186(137)	225(168)	100%(74.6%)

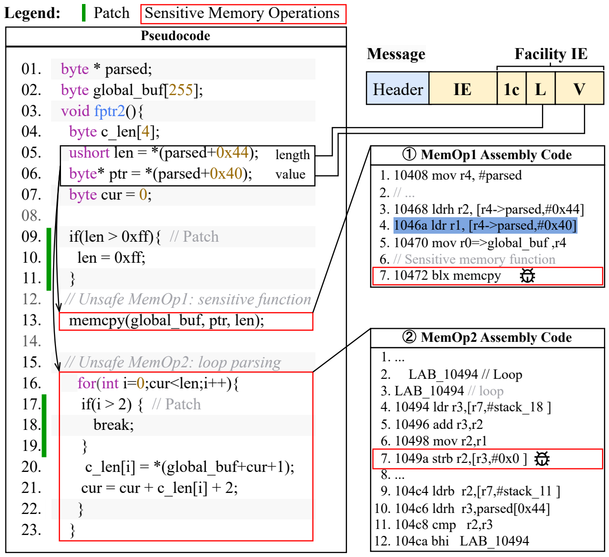
Message-Centric Protocol Implementation For real-time requirements, baseband firmware is usually driven by a real-time operating system (RTOS), such as ShannonOS [15], NucleusOS [47], HLOS [40]. Task is a unit of work or a sequence of instructions that is executed independently in RTOS. Each task in an RTOS has its own execution context, including a stack, entry function and other necessary resources. Vendors commonly divide and implement complex cellular protocols as separate tasks within an RTOS [16, 20, 33]. The main logic of a task in protocol message processing is a message dispatch loop which parses and dispatches the incoming messages.

Baseband firmware utilizes message-centric communication to facilitate interaction between different cellular sub-protocols. As shown in the right part of Figure 1, the message processing flow of the task can be divided as two phases: the common phase managing the message queue (dequeue APIs) and parsing the common structures of incoming messages (decode APIs), and the message specific phase handling the message specific logics (handle APIs). Tasks first receive messages originating on the air interface (OTA message) or sent by other tasks, then pass them to a dedicated handler function for processing based on their message types. In the case that messages are received directly from the air interface, they need to be decoded before being handled.

Threat model In a typical threat model [8] of cellular network, attackers can be mailicious operators or owners of fake base station which can send any malicious OTA message to user equipment. Following this assumption, security researches of baseband firmware usually assume the OTA message is fully attacker-controllable, and the received message will undergo the two processing phases. In this work, we start analysis directly from message specific phase and assume that the message specific contents, *i.e.*, Uniform format, L and V fields, are attacker-controllable. This is feasible since the

Table 3: Detail of the Message-Specific Vulnerabilities

Source	Sensitive Memory Operation	Involved Message Content	Code Involved in Patch	Vendor
CVE-2010-3832 [50]	Sensitive function	Message specific	Message-specific	Qualcomm
CVE-2015-8546 [14]	Sensitive function	Message specific	Message-specific	Samsung
CVE-2018-21090 [5]	Loop parsing	Message specific	Message-specific	Samsung
CVE-2018-14319 [5]	Sensitive function	Message specific	Message-specific	Samsung
Cama-2018 [5]	Loop parsing	Message specific	Message-specific	Samsung
BASESPEC-E6-2021- [29]	Sensitive function	Message specific	Message-specific	Samsung
BASESPEC-2021-E8 [29]	Sensitive function	Message specific	Message-specific	Samsung
BASESPEC-2021-E7 [29]	Sensitive function	Message specific	Message-specific	Samsung
CVE-2020-25279 [20]	Sensitive function	Message specific	Message-specific	Samsung
Charles-2018 [6]	Sensitive function	Message specific	Message-specific	MTK
OffensiveCon-2020-Vul2 [31]	Loop parsing	Message specific	Message-specific	MTK
OffensiveCon-2020-Vul1 [31]	Loop parsing	Message specific	Message-specific	MTK



```
01.  byte * parsed;
02.  byte global_buf[255];
03.  void fptr2(){
04.    byte c_len[4];
05.    ushort len = *(parsed+0x44);
06.    byte* ptr = *(parsed+0x40);
07.    byte cur = 0;
08.
09.    if(len > 0xff){  // Patch
10.      len = 0xff;
11.    }
12.  // Unsafe MemOp1: sensitive function
13.    memcpy(global_buf, ptr, len);
14.
15.  // Unsafe MemOp2: loop parsing
16.      for(int i=0;cur<len;i++){
17.       if(i > 2) {  // Patch
18.         break;
19.       }
20.        c_len[i] = *(global_buf+cur+1);
21.       cur = cur + c_len[i] + 2;
22.      }
23.    }
```

```
1. 10408 mov r4, #parsed
2. // ...
3. 10468 ldrh r2, [r4->parsed,#0x44]
4. 1046a ldr r1, [r4->parsed,#0x40]
5. 10470 mov r0=>global_buf ,r4
6. // Sensitive memory function
7. 10472 blx memcpy
```

```
 1. ...
 2.     LAB_10494 // Loop
 3. LAB_10494 // loop
 4. 10494 ldr r3,[r7,#stack_18 ]
 5. 10496 add r3,r2
 6. 10498 mov r2,r1
 7. 1049a strb r2,[r3,#0x0 ]
 8. ...
 9. 104c4 ldrb  r2,[r7,#stack_11 ]
10. 104c6 ldrh  r3,parsed[0x44]
11. 104c8 cmp   r2,r3
12. 104ca bhi   LAB_10494
```

Figure 2: Example for the Vulnerability Pattern: Attacker-Controllable Input + Sensitive Memory Operations. The example is simplified from a real world case containing two unique vulnerabilities. The left part is the decompiled code which has no precise type information.

common phase treats message specific content as blobs and directly passes them to the message specific phase.

2.2 Empirical Study: Message-Specific Vulnerabilities

To gain insights into the characteristics of the message-specific vulnerabilities, we have conducted a study analyzing all the historical vulnerabilities. We first collected all baseband firmware related vulnerabilities, then identified the message-specific vulnerabilities contained, and finally conducted a reverse-engineering based analysis to understand their root causes and fix patches.

Data Collection The first step involved identifying baseband firmware-related vulnerabilities by performing keyword matching across the security bulletin boards of the top three suppliers, *i.e.*, Qualcomm [41], MediaTek [34], and Samsung [45], over a time span of nine years (2015 - present). A total of 225 vulnerabilities were collected based on vulnerability descriptions containing keywords such as "modem", "baseband", or "cellular". Next step is to filter out the message-specific vulnerabilities based the following criteria:

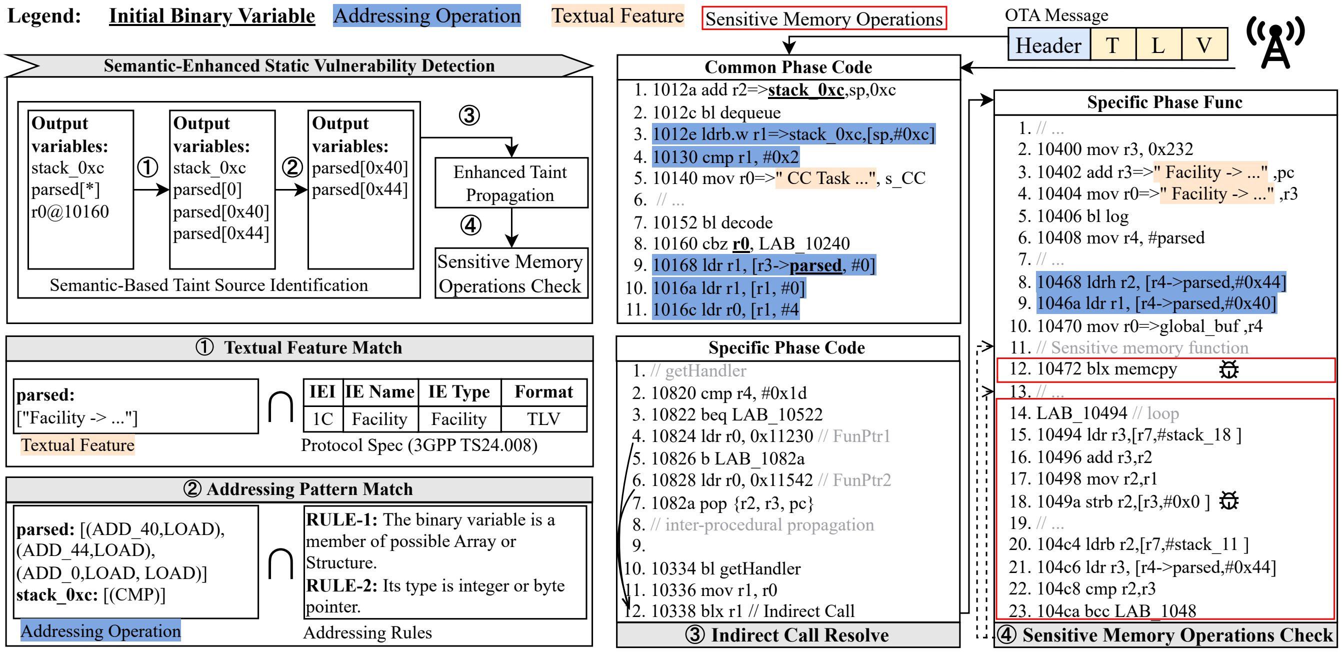
- If the description of a vulnerability declares that it resides in the process of decoding or parsing message, or is related with the general structure of message such as common message header, it is a common phase vulnerability;
- If the description declares that it resides in the message specific processes such as handling specific IE information, authenticating message, or generating message responses, it is a message-specific vulnerability;
- If none of the above criteria were met, the vulnerability is marked as unknown.

Through this process, 95 message-specific vulnerabilities were confirmed. In the final step, we collected their binary-level information and conducted reverse-engineering based analysis for understanding the root causes and fixes of these vulnerabilities. The binary-level information mainly includes the root cause location descriptions and the binary diff of the fix collected from web pages referenced in security bulletin boards, CVE websites, and previous researches [5, 14, 16, 20, 29, 31, 50]. Since there are quite limited public information for baseband vulnerabilities, 12 vulnerabilities are finally successfully analyzed. Table 2 lists the general overview of the collected data while Table 3 provides the detailed statistics of the 12 analyzable message-specific vulnerabilities. Full details of the studied vulnerabilities are listed in [4].

Finding 1: Prevalence As shown in Table 2, in all vulnerabilities whose type can be identified, the amount of message-specific vulnerabilities takes the most proportion (74%, 95/128), which is nearly three times to the amount of common phase vulnerabilities, showing its prevalence. Particularly, around 65% (62/95) of the message-specific vulnerabilities are memory safety issues caused by improper handling of message content.

Finding 2: Straight-Forward Vulnerability Pattern For the 12 analyzable message-specific vulnerabilities, we found that all of them fall into a straightforward detection pattern: attacker controllable input + sensitive memory operations. As shown in Table 3, there are two categories of sensitive memory operations causing these vulnerabilities: ❶ the use of sensitive memory functions and ❷ the intricate loop handling of memory buffers. Figure 2 uses an example to illustrate the pattern. For simplicity, it lists the pseudo code and assemblies simplified from a real world case (Samsung, G960FXXS2BRH8) containing two vulnerabilities (triggered in line 13 and 21 respectively). These two vulnerabilities share the same attacker-controllable input but are caused by different categories of sensitive memory operations. Both vulnerabilities are caused by missing proper checks of the value of parsed IE content.

Finding 3: High Locality Another observation is that all analyzed message-specific vulnerabilities are localized in message-specific phase, which means that the detection or exploitation of these vulnerabilities do not need consider the logic implemented in common phase. The reasons behind this observation is two-fold. On one hand, the two phases process different parts of the message contents due to the essential difference of their tasks. The common phase parses and processes messages based on the common design or structures related with all types of messages while the message-specific phase interprets the content of message-specific fields according to the message type and processes accordingly. In common phase, the message-specific fields, such as L and V fields in L3 message, are treated as blob data and passed through. On the other hand, the message-specific vulnerability type we studied



```
Common Phase Code
 1. 1012a add r2=>stack_0xc,sp,0xc
 2. 1012c bl dequeue
 3. 1012e ldrb.w r1=>stack_0xc,[sp,#0xc]
 4. 10130 cmp r1, #0x2
 5. 10140 mov r0=>" CC Task ...", s_CC
 6.  // ...
 7. 10152 bl decode
 8. 10160 cbz r0, LAB_10240
 9. 10168 ldr r1, [r3->parsed, #0]
10. 1016a ldr r1, [r1, #0]
11. 1016c ldr r0, [r1, #4
```

```
Specific Phase Code
 1. // getHandler
 2. 10820 cmp r4, #0x1d
 3. 10822 beq LAB_10522
 4. 10824 ldr r0, 0x11230 // FunPtr1
 5. 10826 b LAB_1082a
 6. 10828 ldr r0, 0x11542 // FunPtr2
 7. 1082a pop {r2, r3, pc}
 8. // inter-procedural propagation
 9.
10. 10334 bl getHandler
11. 10336 mov r1, r0
12. 10338 blx r1 // Indirect Call
```

```
Specific Phase Func
 1. // ...
 2. 10400 mov r3, 0x232
 3. 10402 add r3=>" Facility -> ..." ,pc
 4. 10404 mov r0=>" Facility -> ..." ,r3
 5. 10406 bl log
 6. 10408 mov r4, #parsed
 7. // ...
 8. 10468 ldrh r2, [r4->parsed,#0x44]
 9. 1046a ldr r1, [r4->parsed,#0x40]
10. 10470 mov r0=>global_buf ,r4
11. // Sensitive memory function
12. 10472 blx memcpy
13. // ...
14. LAB_10494 // loop
15. 10494 ldr r3,[r7,#stack_18 ]
16. 10496 add r3,r2
17. 10498 mov r2,r1
18. 1049a strb r2,[r3,#0x0 ]
19. // ...
20. 104c4 ldrb r2,[r7,#stack_11 ]
21. 104c6 ldr r3, [r4->parsed,#0x44]
22. 104c8 cmp r2,r3
23. 104ca bcc LAB_1048
```

Figure 3: Motivation Example for Semantic-Enhanced Taint Analysis in Static Vulnerability Detection

tends to be localized: detecting or fixing a memory buffer handling issue usually do not involve the whole message process workflow. The third and fourth columns in Table 3 show statistics about the code range involved for patching and the types of inputs responsible for triggering the vulnerabilities, respectively. These results support our finding about the vulnerability's high locality.

Threats to Validity The main threats to validity is the external validity. Due to the limited public information for understanding message-specific vulnerabilities, we can only analyze part of the existing vulnerabilities. The rest vulnerabilities can have different kinds of vulnerability patterns and involve both phases of the message processing. To alleviate that, we collected vulnerability information from multiple sources and conducted manual reverse engineering process to analyze more vulnerabilities. For the internal validity, our manual reverse engineering process or scripts can contain bugs which affects the statistical results.

2.3 Motivation Example

The study has shown clear patterns of the message-specific vulnerabilities, which motivates us to design a static taint analysis based approach to systematically identify similar vulnerabilities in baseband firmware. The basic intuition is straight-forward: we detect potential vulnerabilities by identifying the sensitive memory operations which directly handle the attacker-controllable input. However, to guarantee a highly automatic and precise detection in a complex and close-sourced firmware is challenging. Particularly, two key challenges are identified in both the taint source identification and taint propagation processes. Figure 3 illustrates the summarized technical challenges and our corresponding solutions based on Figure 2's example.

The left-top part of the Figure 3 illustrates a general workflow of the detection while the right part shows the target case in assembly. The two vulnerabilities are marked with bug icons and the assemblies around their root causes, which match the predefined sensitive memory operations (④), are set as the taint sink points. To effectively conduct taint analysis for the detection, the semantic-based taint source identification and enhanced taint propagation are introduced. The former requires precise identification of the binary variables carrying the content of L and V fields of an IE message. Our intuition is to utilize the textual features and addressing patterns of the assemblies to achieve precise location. As shown in example, we first collected an initial set of binary variables representing all outputs of the common phase message processing (`stack_0xc`, all offsets with `parsed`, and `r0@10160`). Then textual features are collected and matched (①) to identify the variables belonging to the certain types of messages, *e.g.*, `parsed[0x44]` and `parsed[0x40]` are tagged as potential taint source due to matched string "`CC Task ..`". Finally addressing pattern match is applied (②) for filtering the variables whose addressing patterns do not behave like a data field of a parsed message, *e.g.*, `stack_0xc/parsed[0]` is filtered by RULE-1/RULE-2 respectively. For the taint propagation, to avoid the early termination caused by unsolved indirect calls, we conducted further analysis to infer their potential callees. The basic intuition here is that the sources of callees for many indirect calls are hardcoded function addresses (`FuncPtr1`, `FuncPtr2`) in binary. Propagating these sources with inter-procedural data flow analysis, some caller-callee mappings can be retrieved. In the example, the callee is returned as `r0` by `getHandler` (③), propagated from switch structure in line 2-6, and finally called at line 12. Based on the above, these two vulnerabilities can be detected.

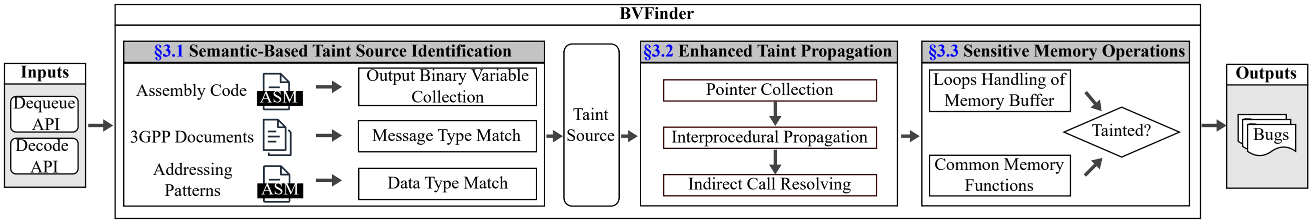


Figure 4: The BVFinder Overview

2.4 Our Approach

Based on the above, we designed and implemented our semantic-enhanced static vulnerability detector BVFinder. As shown in Figure 4, it takes the offsets of common phase functions as input, conducts automatic and precise taint analysis, and outputs the matched potential vulnerabilities. The offsets of common phase functions are assumed as known since identifying them is usually a one-time effort per vendor and there already exist blogs providing detailed reverse-engineering results for all vendors [14, 31]. BVFinder has three key components: ❶ semantic-based taint source identification; ❷ enhanced taint propagation; ❸ and sensitive memory operations. First, it collects all output binary variables of common phase (dequeue and decode APIs) and picks out the variables carrying specific message content via message and data type matches. Second, it conducts an enhanced taint propagation by resolving the indirect calls via an inter-procedural data flow analysis inferring the potential callees. Lastly, vulnerabilities are detected as the tainted sink points. The sink points are the assemblies matching sensitive memory operations summarized from existing vulnerabilities.

3 METHODOLOGY

3.1 Semantic-Based Taint Source Identification

To detect the exploitable message specific phase vulnerabilities, the first step of BVFinder's taint analysis is to precisely identify the taint sources representing attacker-controllable inputs. According to the threat model discussed in Section 2.1, the ideal taint sources should be the L and V fields of the IEs inside certain types of L3 messages. This means that we need to automatically locate the binary variables which contain the data representing specific types of message fields. Our basic observation is that, even in these commercial and closed-source firmware, the assemblies related with the usage of a binary variable still expose enough information for identification. Particularly, the low level data type information for identifying the fields can be inferred from the patterns and semantics of addressing operations in assembly while the high level message type information can be inferred from the textual features contained in log strings and function names. In general, BVFinder first collects all binary variables outputted by common phase as candidates, then applies inference rules to locate the variables which highly match the specific message and data types as taint sources.

Output Binary Variable Collection BVFinder collects all binary variables written by functions from common phase as potential taint sources. As discussed in Section 2.4, the identification of these functions for firmware of mainstream vendors is one-time efforts and already done by the communities. For each function, the output binary variables are return registers, *e.g.*, `r0` in ARM32, stack offsets, *e.g.*, `sp+offset` in ARM32, and global data offsets written by the function. BVFinder traverses and analyzes the assemblies of these functions to collect the output binary variables as initial taint source candidates.

Message Type Match BVFinder leverages textual information contained in the assemblies to infer whether a taint source carries data for a specific message type. The basic intuition is that the function names and log strings contained in message-specific processing functions have the same topic as the message descriptions documented in the specification. Therefore, the more similar the textual features and the descriptions are, the more likely the related binary variable belongs to that type of message. For example, the appearance of debugging information `"Facility -> ... "` may imply that the current code is processing Facility IE of cellular messages. Specifically, as shown in Formula 1, BVFinder uses Jaccard Similarity [25] to conduct the textual feature match. The k in the formula is the similarity threshold, which is empirically determined as 0.8. Section 4.2 analyzes the effects of BVFinder under different k values.

$$\begin{aligned} &Jaccard(x,y) = \frac{|\, x \cap y \,|}{|\, x \cup y \,|} \\ &Matched(v) = MAX(Jaccard(a,b)) > k \\ &a \in textual\ feature, b \in specification, k \in (0,1) \end{aligned} \tag{1}$$

Data Type Match BVFinder does not target on precisely recovering the data type of each binary variable. Instead, it sieves the candidate variables by matching their addressing patterns collected from the assemblies with the expected. Note that ideally the taint sources should only be the L and V fields of an IE message. And their expected addressing patterns should obey the patterns for accessing leaf members of a `struct` or `array`. A leaf member is a member directly carries the data content, which can be either integer types or byte pointer. This expectation comes from our observation on the characteristics of cellular protocol's message-centric design. Since the common phase handles all types of messages but only understands the general structure of a message, it usually parses the incoming message as a general compound structure containing arrays or nested structures. As for message specific content such as L and V fields, it can only treat them as blobs and store them as leaf members in the parsed structure for following processing.

We observe that baseband firmware typically parses messages into composite structures in the form of arrays or structures. For conducting the match, BVFinder first identifies the compound structure, then checks whether the addressing patterns of a candidate taint source behave like a leaf member of the structure. The compound structure is identified based on the specific addressing

```
Algorithm 1: IndirectCallResolving
   Input: an indirect call list $\mathcal{I}$, an ICFG list $icfg$
   Output: a map $\mathcal{M}$ of (k: $i$, v: {}, for $i$ in $\mathcal{I}$)
1  $\mathcal{SP} \leftarrow \{\}$
   // Step-1: Collect potential pointers.
2  foreach $inst \in icfg$ do
3      if isBinOpType(inst) and isSrcOPValidAddr(inst) then
4          $value \leftarrow$ getSrcValue($inst$)
5          $\mathcal{SP} \xleftarrow{+} (inst, value)$
   // Step-2: Taint and propagate.
6  foreach $inst, value \in SP$ do
7      $context \leftarrow$ initTaintCtxt($inst$, $value$)
8      $cur\_inst \leftarrow inst$
9      while hasNext($cur\_inst$, $icfg$) do
10         $cur\_inst \leftarrow$ getNext($icfg$, $cur\_inst$)
11         $context \leftarrow$ doTaintPropagation($cur\_inst$, $context$)
12         if $cur\_inst \notin \mathcal{I}$ then
13             continue
14         $tSrc \leftarrow$ getTaintSources($context$, $cur\_inst$)
           // Step-3: Resolve potential targets.
15         foreach $v \in tSrc$ do
16             $fptr \leftarrow$ getPotentialFuncPtr($v$)
17             if $fptr \neq NULL$ then
18                 $\mathcal{M} \xleftarrow{+} (cur\_inst, fptr)$
19 return $\mathcal{M}$
```

patterns of `struct` or `array`, *i.e.*, adding offsets for accessing its members such as `ldr r0,global, ldr r0,[r0,#offset]`. And a candidate taint source behaves like a leaf member if its addressing patterns involve the base address of the structure (it is a member) and BVFinder cannot find an assembly which uses it as a nested pointer (it is or directly points to data)

3.2 Enhanced Taint Propagation

Another key component of BVFinder is the enhanced taint propagation for these identified taint sources. The motivation for designing this component comes from the observation that the taint propagation of BVFinder is usually aborted due to the abrupt ending of control flow caused by the unsolved indirect calls. After some reverse-engineering efforts to understand the sources of the callees of these indirect calls, we found that the sources of most callees are function addresses hardcoded in binaries rather than some dynamically determined function pointers. These hardcoded function addresses are propagated during the execution, may cross functions, and finally be used by these indirect calls. Specifically, inter-function data propagation can be accomplished by either via function return value, such as returning a hardcoded function address according to the dynamic context using a switch statement, or through global function arrays. We believe this is a common design practice of baseband firmware developers for better management or extension of the hundreds types of message processing code. Existing control flow graph construction algorithms used by popular disassemblers [21, 37, 46] failed to resolve these indirect calls since they only considered intra-procedural data propagations during the CFG construction. Conclusively, to resolve these indirect calls, BVFinder proposed an enhanced call graph construction algorithm which maintains additional data flow propagations of identified hardcoded function addresses.

The algorithm is mainly divided into three steps: ❶ effective pointer collection, ❷ inter-procedural data flow propagation, and ❸ callee solving. Algorithm 1 shows the detailed process. First, BVFinder scans the assembly to collect all effective hardcoded addresses (line 2-5). This is done by collecting and checking the source operands of specific assembly instructions, *e.g.*, `MOV` and `LOAD` in `ARM`. A hardcoded address is effective if it can be interpreted as a pointer pointing to some global data offsets or function starting addresses. Note that non-function addresses are also included in the pointer collection step since it is possible the pointer finally points to a function after following propagation. Then, for each effective pointer, BVFinder uses ❷ and ❸ to find its relation with unsolved indirect calls (line 6-18). BVFinder conducts inter-procedure data propagation to spread the pointer (line 9-18). Once the propagation reaches an unsolved indirect call and the target callee is a concrete value maintained in our propagation, one caller-callee mapping is found (line 15-18). When the data flow reaches the fixed point or reaches the maximum number of iterations, BVFinder finds out all callees of these indirect calls. Based on the above algorithm, BVFinder can conduct the taint propagation more effectively.

3.3 Sensitive Memory Operations

BVFinder summarizes two types of sensitive memory operations for the vulnerability detection. Once the taints reaching a place where assemblies contain the sensitive memory operation patterns and satisfying certain requirements such as some specific arguments are tainted, it will be marked as a potential vulnerability.

The first type of sensitive memory operation is the utilization of common memory functions such as `memcpy`, `strcpy`, etc. These functions have both the `buffer` and `size` parameters. If the value of the `size` parameter is tainted but the length of `buffer` can be smaller than `size`, BVFinder identifies the place as a potential vulnerability. In implementation, to keep the lightweight nature of BVFinder, the latter condition is simplified as checking whether there exists size checks when copying the `buffer`. The second type of operation is the loop handling of memory buffers. In BVFinder a loop is divided as the loop condition and loop body. And the control variables are the variables involved in the loop condition. Vulnerabilities in loops are usually caused by out-of-bounds reads or writes when loop control conditions are affected by external input. Therefore, if a loop control variable is tainted, we identify it as a potential risk location. In implementation, the existence of loops and the loop components are identified based on existing algorithms [27] before the vulnerability detection.

It is possible to build more fine-grained sensitive memory operation patterns using advanced tools such as binary symbolic executors, which can provide a more accurate pattern detection result. However, this raises practical challenges due to the huge performance costs of symbolic execution. BVFinder balances the overall performance and preciseness by using the above lightweight detection patterns.

Table 4: Code Statistics of BVFinder.

Component	LoC
Semantic-Based Taint Source Identification	2606 lines
Enhanced Taint Propagation	1032 lines
Sensitive Memory Operations Check	672 lines
Total	4310 lines

4 EVALUATION

Implementation BVFinder consists of around 4k lines of Java and Python code, as summarized in Table 4. It is built upon several existing libraries or tools. The main modules include data type match, debug information collection and indirect calls resolution, which are implemented on the disassembly engine of `Ghidra` [37]. Since baseband firmware is usually not in standard ELF format, the vendor-specific plugins [13, 19] are used to load and analyze baseband firmware. Further, we utilized standard L3 layer messages specification extracted by BaseSpec [29] for the message type match. We also used `nltk` [3] for string preprocessing and similarity matching. Finally, the inter-procedural taint analysis is built based on BinAbsInspector [27].

Research Questions Our evaluation is designed to tackle the following research questions (RQs):

- **RQ1 (State-of-the-Art Comparison)** - How does BVFinder perform compared with the state-of-the-art tools?
- **RQ2 (Ablation Studies)** - How each component of BVFinder contributes to its vulnerability detection?
- **RQ3 (Real-world Evaluation)** - Can BVFinder find zero-day vulnerabilities?

Configurations We ran all experiments on a host machine equipped with an Intel Core i7-11700 at 2.50 GHz and 128GB of RAM. FirmWire was executed within the Docker environment provided by its website [20]. The Docker was run on the same host machine as BVFinder and each target cellular protocol task is fuzzed for 24 hours.

4.1 RQ1 Comparison with the state-of-the-art

Baselines Two state-of-the-art tools are compared and evaluated to understand the effectiveness of BVFinder. One tool is a static vulnerability detector named BaseSpec which aims to find inconsistency bugs between the cellular specification and the implementation while another tool is a dynamic fuzzer called FirmWire conducting emulation-based fuzz testing towards baseband firmware. The comparison experiment is done by applying these two tools and BVFinder towards old version cellular baseband firmware and comparing their abilities on detecting known vulnerabilities. The known vulnerabilities cover all analyzable ones studied in Table 3 except CVE-2010-3832. This is because that vulnerability is of a firmware using a special ISA (Instruction Set Architecture) called `Hexagon` [3]. which is unsupported by all existing analysis tools.

Evaluated Firmware Since one vulnerability only exists in certain versions of firmware, in total three firmware are used for covering the comparison of all studied vulnerabilities. The three firmware includes G9250CHC1BOGB, G930FXXU1APB4, MT6762. We first collected all released firmware of related vendors from the third-party website [44] and the dataset [12] published by FirmWire, then picked these three which can cover all studied vulnerabilities.

Overall Result Table 5 shows the comparison results. BVFinder shows clear performance advantage by identifying all 11 message-specific vulnerabilities, while BaseSpec and FirmWire only found 3 and 6 vulnerabilities respectively. Note that since BaseSpec's vendor-specific analysis code is not publicly available. Therefore we directly used the results reported in their paper for comparison.

Comparison with BaseSpec BaseSpec is a tool designed to find inconsistency bugs in the common phase. These message-specific vulnerabilities (ID 5,6,7 in Table 5) are not directly related with the inconsistency issues it detected but under these issues' impact scope. Specifically, BaseSpec first located an inconsistent implementation of length check for a specific type of message in common phase, then the user of BaseSpec manually checks whether potential memory safety issues can happen in message specific phase given the implemented check. Indeed, naively fixing the inconsistency in check cannot eliminate these vulnerabilities since the check is designed for ensuring the correct execution of communication rather than the low level memory safety consideration. For the rest vulnerabilities, BaseSpec cannot work since there has no such relations. BVFinder does not have the above limitations since it is an automatic detector designed for message-specific vulnerabilities.

Comparison with FirmWire FirmWire can be used for testing message-specific vulnerabilities. However, it found less than half of the vulnerabilities since it is hard to create complex states required for dynamically reaching these vulnerabilities. Specifically, triggering these vulnerabilities require complex prefix message sequences. For example, ID #7 vulnerability exists in the processing of "Network Daylight Saving Time" IE, triggering it requires the establishment of NAS context. However, this involves a prefix communication sequence containing 10 more messages, which is quite challenging to be created by the coverage-guided fuzzing techniques employed in FirmWire without any guidance of cellular protocol domain knowledge. BVFinder bypasses these requirements since: ❶ it analyzes the firmware statically; ❷ the vulnerabilities it targets logically exist exploitation paths according to the threat model.

Summary to RQ1 Compared with existing vulnerability detectors, BVFinder showed clear performance superiority on detecting message-specific vulnerabilities. This superiority comes from its message-specific design choices on detection.

4.2 RQ2 Ablation Study

Baselines We designed several variants of BVFinder to understand whether and how each component of BVFinder contributes to its overall performance.

- **BVFinder-INIT**, which directly uses all initial output binary variables as taint sources without two match strategies (Section 3.1) and enhanced taint propagation (Section 3.2).
- **BVFinder-STI**, which filters taint sources by applying the two match strategies but does not have enhanced taint propagation.

[3] `Hexagon` is a self-designed ISA of `Qualcomm` [10]

Table 5: Comparison with SOTA Tools

ID	Source	Sensitive Memory Operations	Related IE	Vendor	BASESPEC	FIRMWIRE	BVFINDER
1	CVE-2015-8546 [14]	Sensitive function	Progress indicator	Samsung	○	●	●
2	CVE-2018-21090 [5]	Loop Parsing	Protocol configuration options	Samsung	○	●	●
3	CVE-2018-14319 [5]	Sensitive function	Cause	Samsung	○	●	●
4	Cama-2018 [5]	Loop Parsing	Emergency number list	Samsung	○	○	●
5	BASESPEC-2021-E6 [29]	Sensitive function	P-TMSI	Samsung	●	○	●
6	BASESPEC-2021-E8 [29]	Sensitive function	EMM message	Samsung	●	○	●
7	BASESPEC-2021-E7 [29]	Sensitive function	Network daylight saving time	Samsung	●	○	●
8	CVE-2020-25279 [20]	Sensitive function	Bearer capability	Samsung	○	●	●
9	Charles-2018 [6]	Sensitive function	Bearer capability	MTK	○	○	●
10	OffensiveCon-2020-Vul1 [31]	Loop Parsing	Protocol configuration options	MTK	○	○	●
11	OffensiveCon-2020-Vul2 [31]	Loop Parsing	Emergency number list	MTK	○	○	●

Table 6: Comparison of BVFinder and Its Variants

Approach	Metric	RESULT			Time
		FUNC	LOOP	SUM	
BVFinder-INIT	**TP**	12(9.7%)	0(0%)	12(3.3%)	2712s
	FN	60(83.3%)	2(100%)	62(83.8%)	
BVFinder-STI	**TP**	7(77.8%)	0(0%)	7(9.2%)	984s
	FN	55(76.4%)	2(100%)	57(77.0%)	
BVFinder-ETP	**TP**	72(13.6%)	2(0.2%)	76(4.8%)	1552s
	FN	0(0%)	0(0%)	0(0%)	
BVFinder-ALL	**TP**	**56(95.0%)**	**2(1.9%)**	**58(35.4%)**	1937s
	FN	16(22.2%)	0(0%)	16(21.6%)	

- **BVFinder-ETP**, which uses initial output binary variables as taint sources without the two match strategies but applies enhanced taint propagation.

Evaluated Firmware The firmware G930FXXU1APB4 is used as the evaluation target to cover the most existing vulnerabilities.

Ground Truth Establishment To precisely evaluate the baselines and BVFinder, we established the ground truth data on the selected firmware by conducting a manual validation process. Specifically, we manually marked the union of all detection results (1547) according to the following rules:

- If the reported location is one of the studied vulnerabilities of Table 5, it is a true positive result;
- If a security patch is identified around the location in the binary diff between the firmware and its newer version, it is a true positive result (TP);
- If the above is not satisfied, we validated the location by checking the taint source and the security checks around. This involves the reverse-engineering efforts to gain certain understanding of the logics related. If it does not pass our check, it is a false positive (FP). Otherwise, it is a potential vulnerability and reported to vendor for final validation.

After one man-month mark process, 74 TPs are finally identified. Note that we only seek the help of vendor when a result is highly potential to be zero-day vulnerabilities. During the mark process, we reported seven results to vendors and four of them are TPs.

Overall Result Table 6 shows the comparison results. The FUNC and LOOP columns represent the results of the evaluated tools on two kinds of sensitive memory operations. Generally, the BVFinder-ALL showed the highest TP rate with a decent FN rate while the BVFinder-INIT had lowest TP rate with a high FN rate.

Comparison with BVFinder-STI To understand the contribution of semantic-based taint source identification techniques (abbr as STI), we compared the detection results of BVFinder-INIT with BVFinder-STI, and BVFinder-ETP with BVFinder-ALL. The TP rates of both two comparison pairs have shown a clear increasement while the sheer amount of the TP decreased a little. For example, in FUNC column, the TP number and rate changed from 9.7% (12) to 77.8% (7) by adding STI to BVFinder-INIT and from 13.6% (72) to 95.0% (56) by adding STI to BVFinder-ETP. Other columns showed the same trend. This is reasonable since STI is designed to improve performance by filtering out a more accurate initial taint source set. The rise of TP rate proves the effectiveness of its match strategies while the slight drop of TP number indicates that its overall mismatch is acceptable.

Table 7: TP and FP with different k values

Taint		INITAL	k=**0**	k=**0.2**	k=**0.4**	k=**0.6**	k=**0.8**	k=**1**
Source		1074	136	88	45	20	18	18
TP	**Func**	72(13.6%)	68(22.7%)	68(31.3%)	63(95.5%)	58(95.1%)	56(95.0%)	56(95.0%)
	Loop	2(0.2%)	2(0.3%)	2(0.3%)	2(1.2%)	2(1.3%)	2(1.9%)	2(1.9%)
	SUM	**74(4.8%)**	**70(7.8%)**	**70(8.0%)**	**65(27.8%)**	**60(28.4%)**	**58(35.4%)**	**58(35.4%)**
FN	**Func**	0(0%)	4(5.6%)	4(5.6%)	9(12.5%)	14(19.4%)	16(22.2%)	16(22.2%)
	Loop	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)
	SUM	**0(0%)**	**4(5.4%)**	**4(5.4%)**	**9(12.2%)**	**14(18.9%)**	**16(21.6%)**	**16(21.6%)**

Comparison with BVFinder-ETP Similarly, we analyzed the effectiveness of enhanced taint propagation (abbr as ETP) by comparing the FN number and rate of BVFinder-INIT with BVFinder-ETP, and BVFinder-STI with BVFinder-ALL. The FN rates and sheer amount of both two comparison pairs have shown a clear decreasement. For example, in FUNC column, the FN number and rate changed from 83.3% (60) to 0% (0) by adding ETP to BVFinder-INIT and from 76.4% (55) to 22.2% (16). Other columns have shown the same trend. Since the ETP is designed to provide a more complete CFG to enable the taint propagation on code cannot be analyzed before, these decreasements demonstrate its effectiveness.

Evaluation of Different k Values To understand the best value of k in Formula 1, we further conducted experiments to understand the effectiveness of BVFinder with different k values. Table 7 compares the detection results for six different k values. From the table, we can find that both the TP and FN rate increase when k increases. This leads to a two-fold conclusion on the recommended k value. On one hand, if the user of BVFinder prefers a high TP rate result where minimum manual analysis efforts need to be involved for finding new vulnerabilities, the k should be a high value such as 0.8 or 1. On the other hand, if the user desires a balanced detection result considering both the manual analysis efforts and comprehensive detection of vulnerabilities, 0.4 is a more suitable value.

Table 8: 0-day Vulnerabilities Discovered by BVFinder

CVE-ID	Vul Type	Severity	CVE Status	Last Affected Version
CVE-2023-21503	Buffer overflow	High	Assigned	SMR May-2023 Release 1
CVE-2023-21494	Buffer overflow	High	Assigned	SMR May-2023 Release 1
CVE-2023-21504	Buffer overflow	High	Assigned	SMR May-2023 Release 1
CVE-2022-25821	Out of bound read	Low	Assigned	SMR Mar-2022 Release 1

```
1  void mm_main(){
2      uint msg_id;
3      handler = (code **)(&handler_list)[msg_id];
4      (**handler)();  // handler_list[0xb3] -> DecodeUeCapaReqMsg()
5  }
6  void SendUeCapabilityMsg(){  // called by DecodeUeCapaReqMsg()
7      byte buf[52];
8      byte* ie_ptr = get_ptr();
9      int len = get_len();
10     memcpy(buf,ie_ptr,len); // lack IE length check
11 }
12 void mm_RAURequest(){
13     int len = get_len();
14     byte* ie_ptr = get_ptr();
15     log("MS Radio Access Capability ->"); // string for matching.
16 }
```

Listing 1: Simplified Code of CVE-2023-21503

Summary to RQ2 In BVFinder, semantic-enhanced taint source identification helps in providing more precise detection results while the enhanced taint propagation contribute significantly in avoiding missing vulnerabilities.

4.3 RQ3 Real World Application

Overview BVFinder is applied on the latest real world firmware A505FXXU9-CVG1 (Galaxy A50, Samsung), G973FXXSEFUJ2 (Galaxy S10, Samsung) and A107FXXS8CVK2 (Galaxy A10s, MediaTek) to find new vulnerabilities. With k value 0.8, BVFinder outputted 153 detection results, where 33 is of FUNC and 120 is of LOOP. We manually analyzed these detection results, following the process discussed in ground truth establishment. Part of the previous established ground truth can be resued here to boost the analysis.

In total, four vulnerabilities are found and Table 8 details the findings. All the vulnerabilities have been responsibly reported and fixed. Three of which are high-risk vulnerabilities of buffer overflow and may lead to remote code execution. They have critical security impacts, affecting more than hundreds of millions of mobile devices which use Samsung shannon baseband. Interestingly, during the manual analysis, we additionally found that vendors have introduced specific sanitization functions in firmware in recent years to limit the number of bytes of memory operations when IE copies according to the length constraints of different types of IEs to avoid buffer overflows. By combining IE's type checking sanitization function and comparing it with the specification, multiple inconsistent bugs were found, they were reported and confirmed by vendor. We also applied FirmWire to these firmware and performed 7 days fuzzing using its default setting. However, it cannot find any new vulnerability. After in-depth analysis, we found that the triggering of the vulnerabilities BVFinder found requires prerequisite interactions for fuzzing, and it is difficult for FirmWire to construct such a message sequence to reach the code fragment where the vulnerability is located without manual configuration.

```
1  byte dst[4];
2  void DecodeGmmSimAuthRspMsg(){
3      byte* msg;
4      if(...){ // check state
5          // mm_Authentication1
6          memcpy(dst,(byte *)(msg + 0xc), *(byte *)(msg + 2));
7      } else {
8          // mm_Authentication2
9          memcpy(dst,(byte *)(msg + 0xc), 4);
10     }
11 }
```

Listing 2: Simplified Code of CVE-2023-21494

Case Study 1: MM #1 We found a buffer overflow vulnerability in mm task. This stack-based buffer overflow vulnerability we found in cellular baseband firmware is related to MS Radio Access capability IE in GPRS Mobility Management Messages. The baseband firmware allocates a fixed stack space for the destination buffer according to the maximum value specified by the cellular protocol specification. Due to the lack of checking of its length field when copying the content of the IE, and the content and length of the IE may be controlled by attacker, it may cause a stack-based buffer overflow. After solving the indirect call in line 4, the semantic-based taint source identification module infers that the variable `len` and `ie_ptr` are related to MS Radio Access capability IE based on the debugging information on line 15 and then marks them as a taint source. Finally BVFinder reports this according to our specified behavior of sensitive memory operations.

Case Study 2: MM #2 We discovered another buffer overflow vulnerability in the authentication process of the GMM protocol. The vulnerability is caused by a lack of checking the length of the Signed Response (SRES) in the Authentication Response IE during the generation of the corresponding message. We confirmed the vulnerability at line 6 by examining a code snippet with the same functionality at line 9. For this vulnerability, an emulation-based dynamic detection technique needs to support a complete forensic process and construct a specific sequence of messages to be executed to the branch where the vulnerability is located.

Summary to RQ3 BVFinder can be used for locating high security impact zero-day vulnerabilities in baseband firmware. Its high TP rate facilitates the real-world detection scenario.

5 DISCUSSION

False Positives False positives occur when a detector incorrectly marks non-vulnerable code as vulnerable. This typically happens when detectors fail to account for certain control flow conditions. In the context of BVFinder, the omission of control flow guard conditions largely results from two types of inaccuracies in the detection process. Firstly, our vulnerability detection patterns are empirically summarized, which is not infallible. For instance, when applying loop-related patterns, BVFinder focuses only on the loop control variables. However, the loop body may include necessary checks that BVFinder does not identify. Secondly, inherent limitations in static analysis can lead to false positives. Specifically, for certain memory operations, it is hard to completely collect their guard conditions. This is because the checks may reside in code areas far from the locations of these memory operations. Precisely identifying the connections between the checks and these memory operations can involve the resolution of indirect calls and pointers, which is challenging in static analysis, especially when analyzing these baseband binaries where source code is not available.

Towards More Baseband Security Analysis A main technical part of BVFinder improves the static taint analysis ability of baseband firmware. By properly leveraging the domain knowledge, it recognizes the attacker controllable input and builds a more precise CFG, providing a more accurate input reachability analysis result. The reachability information provides the semantics of certain binary variables inside the firmware, which is fundamental for many baseband firmware program analysis. For example, besides detecting memory safety vulnerabilities, the information can benefit all other kinds of static vulnerability detection techniques, such as the symbolic execution. It may also be useful in other security analysis scenarios such as binary patch analysis, binary similarity based applications, etc.

Detecting More Types of Vulnerabilities In this work, we mainly focused on detecting memory-safety vulnerabilities in message-specific phase. However, as shown in Section 2.2, 34.7% (33/95) of vulnerabilities are of other kinds of vulnerabilities such as message replay [18, 22, 26], downgrade [23, 43], IMSI catching [38], etc. These vulnerabilities are logical issues of the implementation and detecting them requires the design of advanced detection methods based on the in-depth semantic understanding of the protocol interactions. Though BVFinder currently is not aware of this kinds of logical issues, its precise taint analysis ability and the exploitation of usable domain knowledge inside the firmware binary can lay a better foundation for building advanced solutions.

6 RELATED WORK

Cellular Baseband Security Early research by researchers [35, 36]on cellular baseband focused on fuzzing SMS or broadcast messages from cellular devices.Subsequent work [50] constructs air interface messages through software-defined radios, exploiting baseband firmware corruption vulnerabilities for remote attacks. Golde et al. [14] and Cama [5] analyzed recent Exynos firmware and discovered RCE 0-days and were rewarded at Mobile Pwn2Own. Most of these works are based on manual analysis. BaseSpec [29] and BaseComp [28] perform a comparative analysis of baseband software and cellular specifications. They systematically inspect message structures or integrity protection function to find potential vulnerabilities in the message processing phase.

Cellular Protocols Security Numerous existing research works use formal analysis methods to find design vulnerabilities of cellular protocols by analyzing protocol specification documents, such as CNetVerifier [48], LTEInspector [22], 5GReasoner [23], etc. These works aim to model L3 layer protocols in cellular specifications using expert experience and then use model checking tools to identify security violations. The protocols are analyzed to ensure that they meet the required security properties. In addition, there are some researches based on black-box testing, by constructing test cases and monitoring the abnormal behavior of the device under test through log information to analyze the implementation of the cellular protocol stack of the device. LTEFuzz[30] utilize dynamic testing technology to analyze and extract domain knowledge related to security attributes and expected security goals in the specification. They construct counterexamples and monitor the execution of test cases to verify whether the corresponding specification violates the expected security goal of the protocol design. This approach provides a practical method for verifying the security of protocol designs and identifying potential security vulnerabilities.Recent work [8] attempts to automate document analysis and tests LTE NAS layer protocol stack implementations.

Firmware Static Analysis Static taint analysis is widely used to detect bugs or vulnerabilities in various programs [1, 2, 11, 32, 49] including firmware binaries [7, 9, 42]. The problem of statically locating taint sources and finding targets of indirect calls in stripped binaries is challenging. Existing studies such as SatC [7] and Karonte [42] are mainly aimed at IOT firmware. The taint sources of these firmwares are usually introduced by standard library functions (linux-based system), and their taint source identification strategies are not suitable for statically compiled baseband firmware. Binary analysis tools, such as Ghidra or angr, only identify targets based on constant propagation within current function or basic block, making them miss many indirect call targets. BVFinder utilizes inter-procedural data flow analysis to collect potential function pointers, allowing us to analyze more indirect calls to target functions.

7 CONCLUSION

In this work, we designed a semantic-enhanced static vulnerability detector which focuses on detecting message-specific vulnerabilities. In general, it detects the vulnerabilities by first identifying the binary variables representing attacker-controllable input, then adopting static taint analysis to check whether the input can reach predefined sensitive memory operations. Specifically, two techniques are proposed for enhancing both the automation and effectiveness of the detector. The evaluation demonstrates that BVFinder can detect all existing vulnerabilities in our dataset, which clearly outperforms the state-of-the-art detectors. Based on BVFinder, four high security impact vulnerabilities are found with four CVEs and 12,410 USD bug bounty assigned. Three of them can cause potential remote code execution, affecting hundreds of millions of end devices.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable feedback on an earlier version of this paper. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103900 and Chinese National Natural Science Foundation (Grants #62032010, #62202462, #62302500), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), NRF Investigatorship NRF-NRFI06-2020-0001, National Research Foundation Singapore and DSO National Laboratories under the AISingapore Programme (AISG Award No: AISG2-RP-2020-019). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation of Singapore, Cyber Security Agency of Singapore, and DSO National Laboratories.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency {Use-After-Free} bugs in linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 255–268.

[3] Edward Loper Bird, Steven and Ewan Klein. 2009. Natural Language Processing with Python. O'Reilly Media Inc. https://github.com/nltk/nltk.

[4] BVFinder. 2023. BVFinder-Data. https://sites.google.com/view/bvfinder-data.

[5] A. Cama. 2018. A walk with Shannon.

[6] Anna Dorfman Charles Muiruri, Nitay Artenstein. 2018. The Baseband Basics: Understanding, Debugging and Attacking the Mediatek Communication Processor. https://kenya.opcde.com/speakers.html.

[7] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 303–319.

[8] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. 2021. Bookworm Game: Automatic Discovery of LTE Vulnerabilities Through Documentation Analysis. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1197–1214. https://doi.org/10.1109/SP40001.2021.00104

[9] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 430–441.

[10] Lucian Codrescu et al. 2013. Qualcomm Hexagon DSP: An architecture optimized for mobile multimedia and communications.. In *Hot Chips Symposium*. 1–23.

[11] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. {FIRMSCOPE}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware. In *29th USENIX security symposium (USENIX Security 20)*. 2379–2396.

[12] FIRMWIRE. 2022. FIRMWIRE-Dataset. https://zenodo.org/record/6516030/.

[13] FirmWire. 2022. FirmWire-Ghidra. https://github.com/FirmWire/ghidra.

[14] Nico Golde and Daniel Komaromy. 2016. Breaking Band: reverse engineering and exploiting the shannon baseband.

[15] Marius Muench Grant Hernandez. 2020. Reversing & Emulating Samsung's Shannon Baseband. https://hardwear.io/netherlands-2020/presentation/samsung-baseband-hardwear-io-nl-2020.pdf.

[16] Marco Grassi, Muqing Liu, and Tianyi Xie. 2018. Exploitation of a modern smartphone baseband. *Black Hat USA* (2018).

[17] GSMA. 2022. The mobile economy. https://www.gsma.com/mobileeconomy/wp-content/uploads/2022/02/280222-The-Mobile-Economy-2022.pdf.

[18] Changhee Hahn, Hyunsoo Kwon, Daeyoung Kim, Kyungtae Kang, and Junbeom Hur. 2014. A privacy threat in 4th generation mobile telephony and its countermeasure. In *Wireless Algorithms, Systems, and Applications: 9th International Conference, WASA 2014, Harbin, China, June 23-25, 2014. Proceedings 9*. Springer, 624–635.

[19] Grant Hernandez. 2020. ShannonBaseband. https://github.com/grant-h/ShannonBaseband.

[20] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. 2022. FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware. In *Network and Distributed Systems Security Symposium (NDSS) 2022*.

[21] Hex-Rays. [n. d.]. IDA PRO. https://www.hex-rays.com/products/ida..

[22] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_02A-3_Hussain_paper.pdf

[23] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 2019. 5GReasoner: A property-directed security and privacy analysis framework for 5G cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 669–684.

[24] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. 2021. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1082–1099.

[25] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[26] Imtiaz Karim, Syed Rafiul Hussain, and Elisa Bertino. 2021. 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, 773–785.

[27] KeenSecurityLab. 2022. BinAbsInspector. https://github.com/KeenSecurityLab/BinAbsInspector.

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

[29] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. 2021. BaseSpec: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols.. In *NDSS*.

[30] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. 2019. Touching the untouchables: Dynamic security analysis of the LTE control plane. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1153–1168.

[31] Kira M. Grassi. 2020. Exploring the MediaTek baseband. https://speakerdeck.com/marcograss/exploring-the-mediatek-baseband.

[32] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. 1007–1024.

[33] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. Basesafe: Baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM conference on security and privacy in wireless and mobile networks*. 122–132.

[34] Mediatek. 2023. Product Security Bulletin. https://corp.mediatek.com/product-security-bulletin.

[35] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale.. In *USENIX Security Symposium*, Vol. 168. San Francisco, CA.

[36] Collin Mulliner and Charlie Miller. 2009. Fuzzing the Phone in your Phone. *Black Hat USA* 25 (2009), 31.

[37] NSA. 2019. Ghidra. https://github.com/NationalSecurityAgency/ghidra.

[38] Ivan Palamà, Francesco Gringoli, Giuseppe Bianchi, and Nicola Blefari-Melazzi. 2021. IMSI Catchers in the wild: A real world 4G/5G assessment. *Computer Networks* 194 (2021), 108137.

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

[40] Qualcomm. 2023. Hexagon DSP SDK. https://developer.qualcomm.com/forums/software/hexagon-dsp-hlos.

[41] Qualcomm. 2023. Qualcomm Product Security. https://www.qualcomm.com/company/product-security.

[42] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1544–1561.

[43] David Rupprecht, Kai Jansen, and Christina Pöpper. 2016. Putting {LTE} security functions to the test: A framework to evaluate implementation correctness. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*.

[44] SamMobile. 2023. Download firmware updates for your Samsung mobile phone and tablet. https://sammobile.com/firmwares

[45] Samsungmobile. 2023. Samsung Mobile Security. https://security.samsungmobile.com/main.smsb.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 138–157.

[47] Siemens. 2023. Nucleus RTOS. https://www.plm.automation.siemens.com/global/en/products/embedded/nucleus-rtos.html.

[48] Guan-Hua Tu, Yuanjie Li, Chunyi Peng, Chi-Yu Li, and Songwu Lu. 2015. Detecting problematic control-plane protocol interactions in mobile networks. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 1209–1222.

[49] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.

[50] Ralf-Philipp Weinmann. 2012. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks.. In *WOOT*. 12–21.

[51] Project Zero. 2023. Multiple Internet to Baseband Remote Code Execution Vulnerabilities in Exynos Modems. https://googleprojectzero.blogspot.com/2023/03/multiple-internet-to-baseband-remote-rce.html.