École Polytechnique Fédérale de Lausanne

# Analysis of Type-Confusion Bugs in Trusted Applications

by Károly Artúr Papp

Bachelor Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. Marcel Busch
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 13, 2022

# Acknowledgments

I would like to thank Marcel Busch for his help and pedagogical approach during the course of this project and Deborah Kiszely-Papp for proofreading the report.

*Lausanne, June 13, 2022* Károly Artúr Papp

## Abstract

Most modern-day smartphones use a Trusted Execution Environment (TEE), which provides safety for certain tasks, e.g., authentication and payment processing. The TEE contains Trusted Applications (TAs), which deal with individual tasks on behalf of the TEE. TEEGRIS, the TEE developed by Samsung themselves, was deployed on the S10 in March 2019 and since then has been the TEE used on devices with Exynos chipsets. A type-confusion bug was found, wherein the parameters passed to a TA could be interpreted as a memory reference, even if they were not meant as such. A malicious actor could thus potentially access and overwrite sensitive data. Due to the large number of phones sold with TEEGRIS, this bug could potentially impact millions of users. In this project we examined the extent of this vulnerability, and we have found that on the Samsung S10, 26-29% of earlier TAs (pre-2019/07) are vulnerable.

## Introduction

With the expansion of tasks performed by smartphones, the need has arisen for tools that provide security guarantees. One of these tools, Trusted Execution Environments (TEE) makes use of Trusted Applications (TA) to complete individual tasks. TEEGRIS, the proprietary TEE of Samsung, provides API documentation to developers who write the TAs, in which they detail the necessary interfaces a TA must have. Some functions in a TA take parameters which, depending on a second parameter, may be interpreted as values or memory references. When working with the API provided by Samsung, the developers themselves must check that the parameters are used as intended. As with all human work, this is prone to error and thus it is our opinion that this system is not sufficiently safe. The job of a TEE is to guarantee security, and such a vulnerability may undermine that.

Samsung sold more than 295 million smartphones in 2019 alone, 37 million of which were S10 devices. All S10s purchased outside the US were shipped with an Exynos chipset. The Korean company has continued to deploy them ever since, with Exynos reaching a market share of 14.1% in 2020. Taking into consideration that more than 1.3 Billion smartphones were sold that year, this shows that any potential security vulnerabilities may have global repercussions. It is therefore a necessary task to assess the extent of such vulnerabilities.

During the course of this project 1750 TA functions were analyzed, which constitutes a comprehensive overlook of the firmware images available for the S10, from 2019 to present. Our findings showed that indeed, one may find vulnerable TAs in the earlier S10 firmware images, but none after July, 2019. In some of the vulnerable TAs the type of parameters was checked, but the result did not change the flow of the program. This can be interpreted as obliviousness to the vulnerability examined in this project.

In light of our results, one may conclude that further large-scale analysis is necessary, and we are of the opinion that the paradigm of trusting developers to implement the necessary checks for mitigating this type of bug is insufficient in terms of security.

## Background

The following paragraph is based on an article published on riscure.com [7, 8], since it gives a comprehensive background explanation of the problem we are examining.

The TEE aims to separate the handling of potentially sensitive information from the Rich Execution Environment (REE), the operating system running on the device and accessible to the end user. When the REE needs to process a payment, authenticate a user, or encrypt data it makes a request towards the TEE to perform the necessary operations. The TEE then delegates the given task to a TA. To guarantee separation, the TAs possess a lower level of security clearance (EL0) than the TEE OS (EL1), the operating system responsible for communicating between the TAs and the REE (Figure 1). TAs can either be loaded at initialization or at runtime.

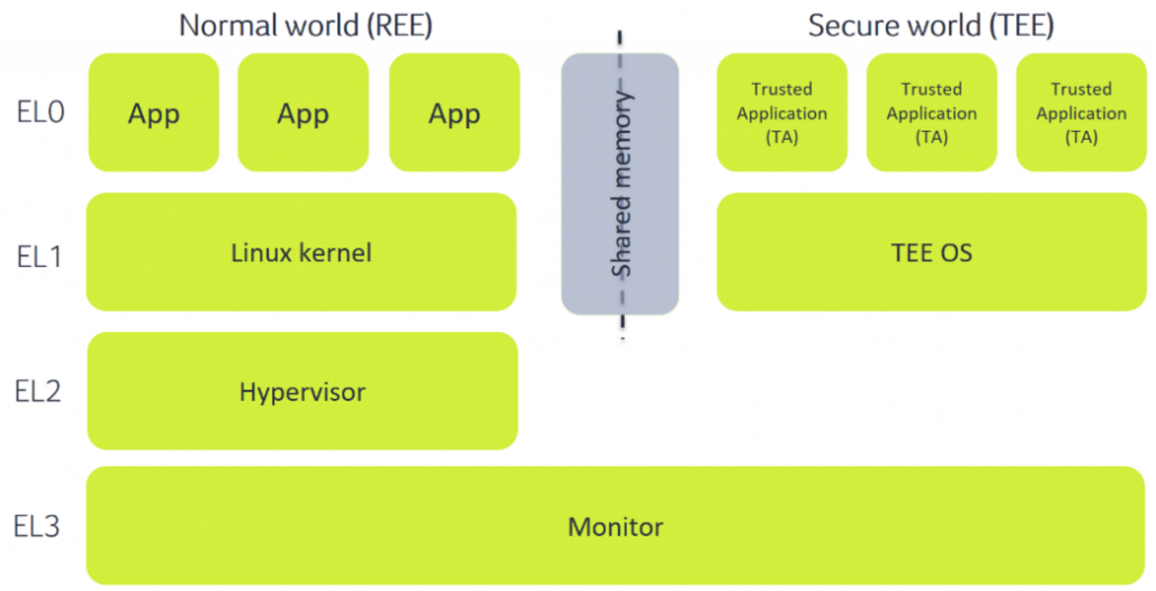
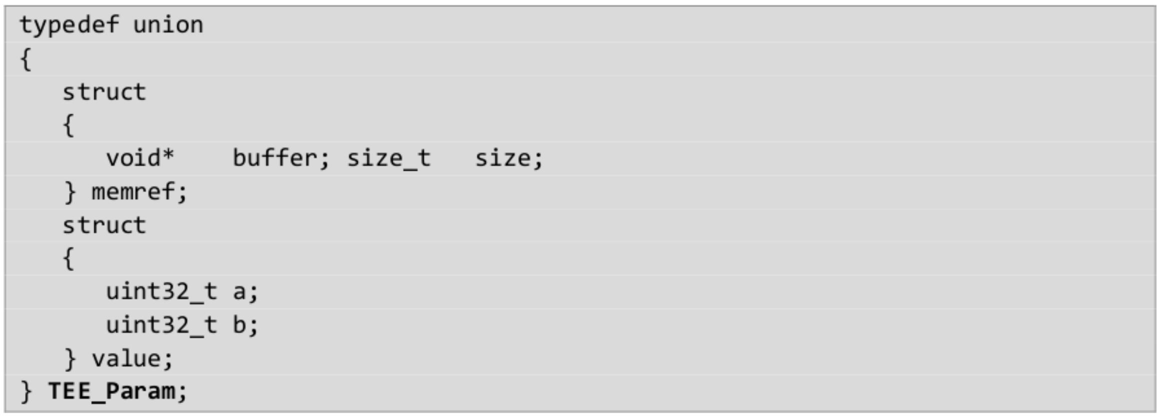


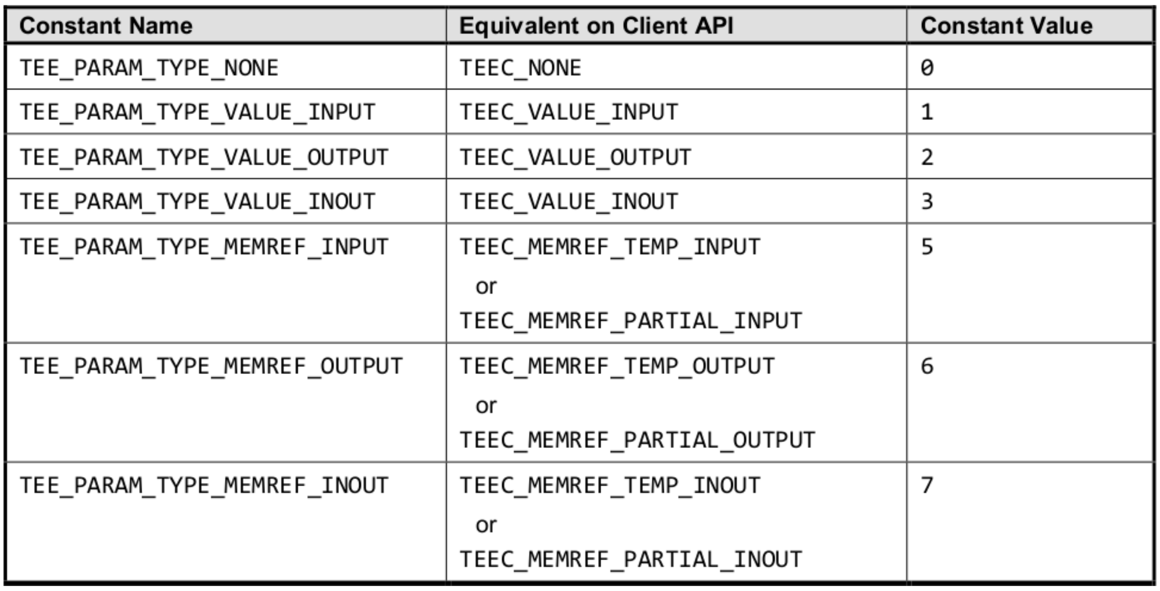
Figure 1: Structure of the communication between the TEE and the REE [7]

TEEGRIS, the TEE examined in this project, was introduced on the S10 device in March 2019, and has been in use since then. The TAs in TEEGRIS get input in the form of a `TEE_Param` (Figure 2), a union which can either be a `value` or a `memref`. To decide between these two options, and to get further details, the functions that are passed to these `TEE_Params` also take a `uint32_t` called `paramTypes`, the potential values of which are detailed below (Figure 3). The particularity of the `union` is that it can be of either type. In our case this means that a `TEE_Param` can be either of type `memref` or of type `value`. The programmer writing the TA is responsible for assuring that the `TEE_Param` is used as intended.



```
typedef union
{
   struct
   {
      void*    buffer; size_t   size;
   } memref;
   struct
   {
      uint32_t a;
      uint32_t b;
   } value;
} TEE_Param;
```

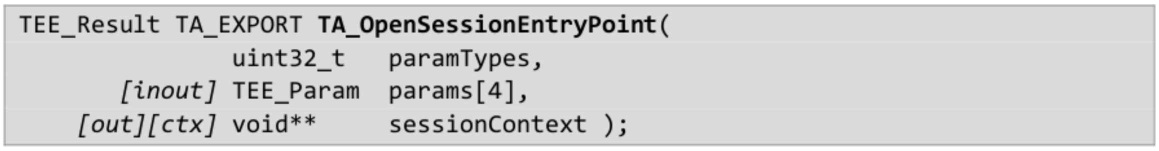
Figure 2: The `TEE_Param` union as per the GP API description [5]



| Constant Name | Equivalent on Client API | Constant Value |
|---|---|---|
| TEE_PARAM_TYPE_NONE | TEEC_NONE | 0 |
| TEE_PARAM_TYPE_VALUE_INPUT | TEEC_VALUE_INPUT | 1 |
| TEE_PARAM_TYPE_VALUE_OUTPUT | TEEC_VALUE_OUTPUT | 2 |
| TEE_PARAM_TYPE_VALUE_INOUT | TEEC_VALUE_INOUT | 3 |
| TEE_PARAM_TYPE_MEMREF_INPUT | TEEC_MEMREF_TEMP_INPUT or TEEC_MEMREF_PARTIAL_INPUT | 5 |
| TEE_PARAM_TYPE_MEMREF_OUTPUT | TEEC_MEMREF_TEMP_OUTPUT or TEEC_MEMREF_PARTIAL_OUTPUT | 6 |
| TEE_PARAM_TYPE_MEMREF_INOUT | TEEC_MEMREF_TEMP_INOUT or TEEC_MEMREF_PARTIAL_INOUT | 7 |

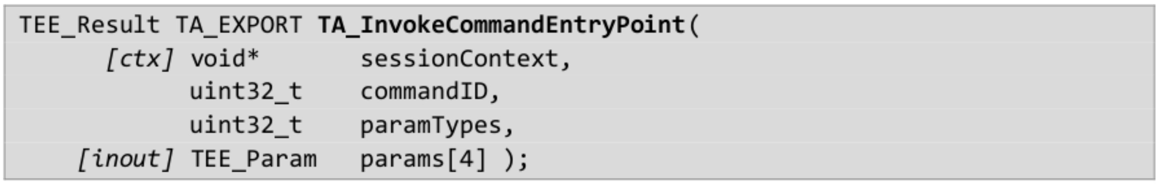
Figure 3: The values the `paramtype` can take, and their meaning as per the GP API description [5]

The developers who write the TAS are provided with the documentation of the TEE Internal Core API Specification [5], which details the necessary functions a TA must contain. A TA must contain 5 functions, of which we will focus on 2 in this paper: 1) TA_OpenSessionEntryPoint (Figure 4), used when an application in the REE requests a connection to the TA, and 2) TA_InvokeCommandEntryPoint (Figure 5), which implements the main functionality of the TA. As we can see both contain an array of `TEE_Params` and a `paramTypes` value.



```
TEE_Result TA_EXPORT TA_OpenSessionEntryPoint(
                uint32_t   paramTypes,
        [inout] TEE_Param  params[4],
     [out][ctx] void**     sessionContext );
```

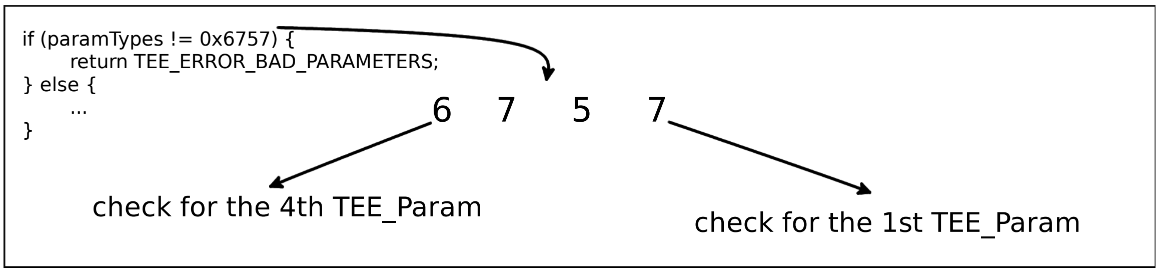
Figure 4: Function header of TA_OpenSessionEntryPoint as per the API description [5]



```
TEE_Result TA_EXPORT TA_InvokeCommandEntryPoint(
      [ctx] void*       sessionContext,
            uint32_t    commandID,
            uint32_t    paramTypes,
    [inout] TEE_Param   params[4] );
```

Figure 5: Function header of TA_InvokeCommandEntryPoint as per the API description [5]

The check is performed in the following way: The least significant 16 bits of `paramTypes` are divided into 4 and each nibble (unit of 4 bits) represents the `paramType` for one `TEE_Param` in the way shown below (Figure 6).

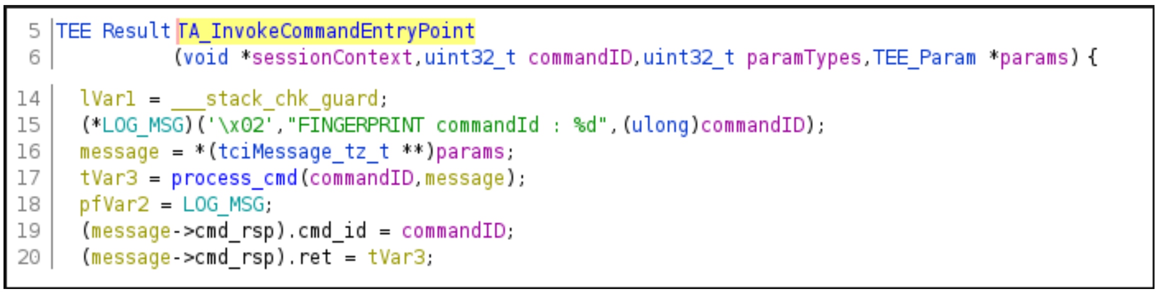


```
if (paramTypes != 0x6757) {
      return TEE_ERROR_BAD_PARAMETERS;
} else {
      ...
}
```

Figure 6: Detailed view of how the check is performed

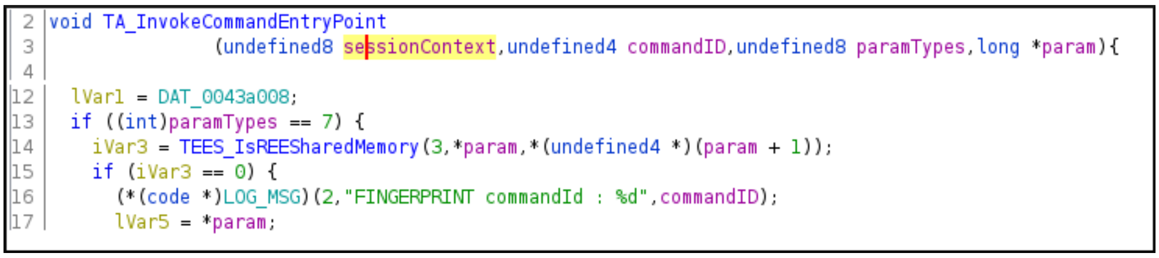
The vulnerability discussed in this report is a prime example of a type-confusion bug caused by the `union`. If the aforementioned check is not performed, an attacker could achieve read and write capabilities and thus gain runtime control of the TA.

To further illustrate this point, here is an example of a vulnerable TA (Figure 7) and a later version of the same TA, where proper checks are carried out Figure(Figure 8). As we see in Figure 7, the first `TEE_Param` is used without any previous check. On the contrary, the check shown in Figure 8 ensures that the `TEE_Param` accessed is of the correct type.



```
TEE Result TA_InvokeCommandEntryPoint
          (void *sessionContext,uint32_t commandID,uint32_t paramTypes,TEE_Param *params) {

  lVar1 = ___stack_chk_guard;
  (*LOG_MSG)('\x02',"FINGERPRINT commandId : %d",(ulong)commandID);
  message = *(tciMessage_tz_t **)params;
  tVar3 = process_cmd(commandID,message);
  pfVar2 = LOG_MSG;
  (message->cmd_rsp).cmd_id = commandID;
  (message->cmd_rsp).ret = tVar3;
```

Figure 7: Vulnerable TA_InvokeCommandEntryPoint from TA 00000000-0000-0000-0000-46494e474502 from firmware version G973FXXU1ASBA (Ghidra screenshot)



```
void TA_InvokeCommandEntryPoint
               (undefined8 sessionContext,undefined4 commandID,undefined8 paramTypes,long *param){

  lVar1 = DAT_0043a008;
  if ((int)paramTypes == 7) {
    iVar3 = TEES_IsREESharedMemory(3,*param,*(undefined4 *)(param + 1));
    if (iVar3 == 0) {
      (*(code *)LOG_MSG)(2,"FINGERPRINT commandId : %d",commandID);
      lVar5 = *param;
```

Figure 8: Safe TA_InvokeCommandEntryPoint from TA 00000000-0000-0000-0000-46494e474502 from firmware version G973FXXU3ASJD (Ghidra screenshot)

## Analysis

For this work, we chose to concentrate on the S10 device, since it was the first to use TEEGRIS, and thus one has access to firmware images that date back to February, 2019. As of the writing of this report, there are 44 firmware images available for the S10, each containing 33-34 TAs, which amounts to  1450 TAs per region that one may examine. Samsung provides firmware images for the S10 in 191 regions, which amounts to  276950 TAs. To analyze that many TAs manually is not feasible, so we made use of a static analyzer, developed in a previous HexHive project and further refined by this author. Even so, and given the fact that the analysis of one TA takes  1 minute, we resorted to further reducing the data sample, the method of which is detailed in the Evaluation section.

The following paragraph is a brief, high-level description of the static analyzer. Ghidra disassembles the binary file, the first block is accessed, and its successors are located. After analyzing the PCODE in the block, it saves state and chooses which block to access next. Calls are prioritized, since these cannot end the function. The function is then called recursively on the blocks, resulting in a DFS-style tree traversal. While doing the analysis the algorithm marks certain blocks as safe, and does not access them. As seen in Figure 9, the control graph keeps track of safe and unsafe branches. In the latter, use of parameters is signaled.

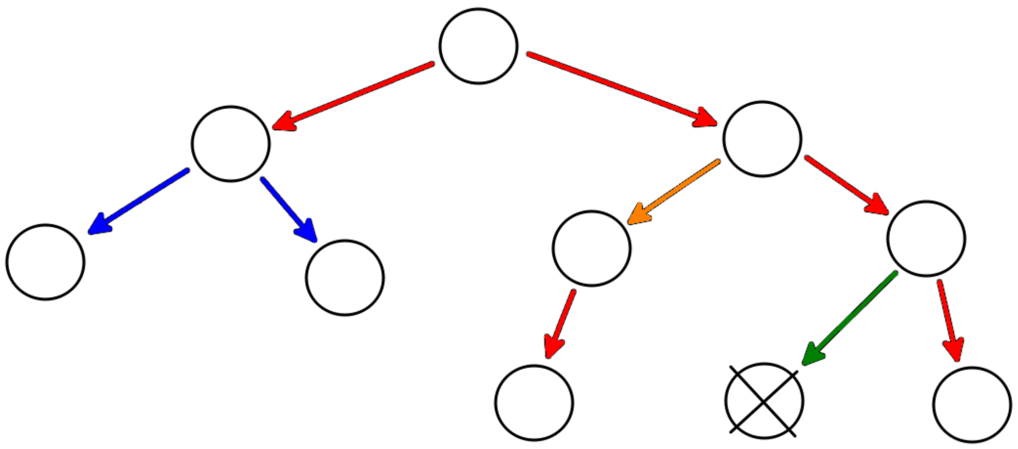


Figure 9: High-level description of control flow of the static analyzer. Red edges signify that checks have not been performed, `param` accesses are potentially unsafe. Blue edges mean that checks were done, `param` accesses are safe. Orange edges mean unsafe function calls (with unchecked parameters), and green edges signify proven safe function calls, which are not pursued further, to reduce costs

A large part of my work on this project was done on the static analyzer. Instead of just outputting to the terminal, the analysis now gets saved in an individual json file for each TA, which provides the address, the type, the affected parameters and a short descriptive text for each error found by the static analyzer. With the help of this output, a "test suite" was made to help future work on the static analyzer. Here is a non-comprehensive list of the bugs that required fixing:

- The program was not making sure that the checked number was actually valid, meaning a check with 0x4 would also contribute to a parameter being declared as safe, even though it might not be.
- Often the value against which the `paramType` gets checked is moved, and this did not always get tracked.
- Often the `params` are moved, but this did not always get tracked.
- `paramTypes` is sometimes masked and/or shifted, but this was not properly tracked.
- The control flow was not optimal, which led to some unnecessarily long runtimes and/or memory faults.

These bugs were fixed, but the static analyzer still has certain limitations, which will be detailed in the Discussion section.

The TAs were analyzed in the following fashion: All TAs in a firmware image were analyzed with the static analyzer. Five of the earlier images were also checked manually with Ghidra in their entirety. This was to ensure that the static analyzer functions as expected, and also because we were aware that the earlier firmware images had a higher chance of containing vulnerable TAs. The static analyzer was tweaked to avoid false negatives as much as possible. After the first 7 images were analyzed, we took the md5 hash of each TA and compared them to previous versions. If they had the same md5 hash, running the static analyzer was not necessary. The TAs marked vulnerable after the 7th image (Version code G973FXXU3ASG8 ) were checked manually with Ghidra. For further large-scale research it may be beneficial to perform other similarity checks.

## Evaluation

Parallel research at the HexHive lab concluded that TAs do not differ over regions. During my research I found that the vulnerability status of TAs does not differ if the TAs have the same SW REV number. Thus we took at least one firmware image for each SW REV code. This resulted in a total of 900 TAs, ranging from the very first firmware image(Version code G973FXXU1ASBA, built on 2019/02/25), to Version code G973FXXUEHVC6, built on 2022/03/10.

During the course of our research, we found that some TAs are not present in all of the firmware versions (Figure 10), but this represents only 9 out of the 35 different TAs that were examined. To our surprise we also found a TA (00000000-0000-0000-0000-505256544545), that did not contain any executable code.

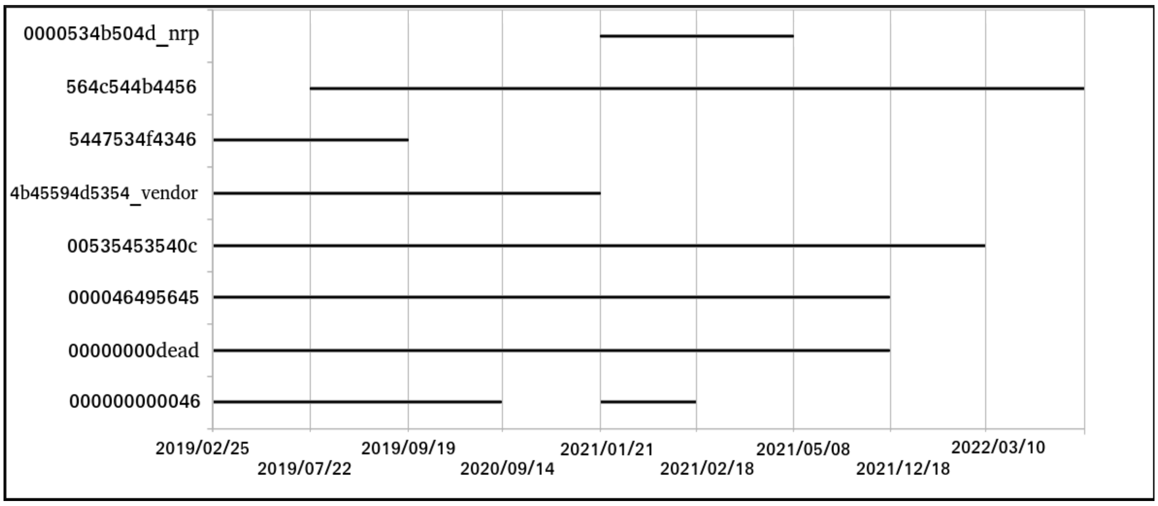
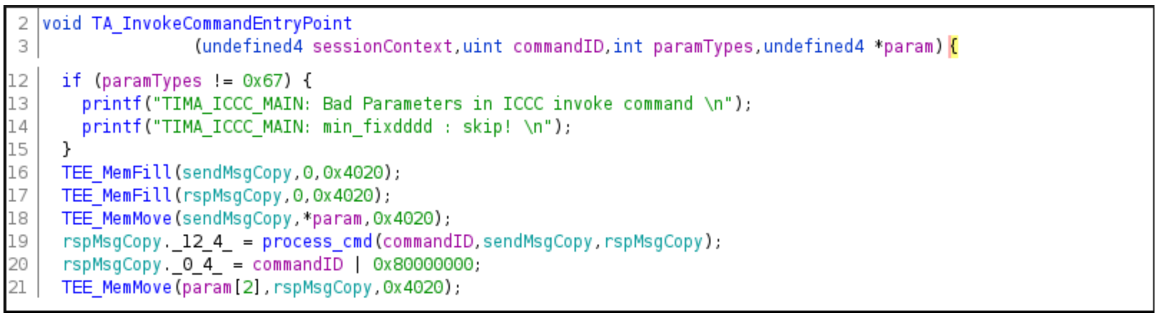


Figure 10: Timeline of TAs not present in all the firmware images

In the firmware images with SW REV codes 1 and 2, we found some glaring vulnerabilities, most alarming was those ones present in TAs 00000000-0000-0000-0000-0053545354ab and 00000000-0000-0000-0000-5354494d4552. In these versions the check itself was not omitted, but nevertheless, the program continues after printing an error message (Figure 11). Unfortunately, we must conclude that this stems from the lack of awareness on the part of the developers regarding this particular bug. This in turn further prvoes the hypothesis that leaving such a check to developers is questionable from a security perspective.



```
void TA_InvokeCommandEntryPoint
               (undefined4 sessionContext,uint commandID,int paramTypes,undefined4 *param){

  if (paramTypes != 0x67) {
    printf("TIMA_ICCC_MAIN: Bad Parameters in ICCC invoke command \n");
    printf("TIMA_ICCC_MAIN: min_fixdddd : skip! \n");
  }
  TEE_MemFill(sendMsgCopy,0,0x4020);
  TEE_MemFill(rspMsgCopy,0,0x4020);
  TEE_MemMove(sendMsgCopy,*param,0x4020);
  rspMsgCopy._12_4_ = process_cmd(commandID,sendMsgCopy,rspMsgCopy);
  rspMsgCopy._0_4_ = commandID | 0x80000000;
  TEE_MemMove(param[2],rspMsgCopy,0x4020);
```

Figure 11: Checked, but still vulnerable TA_InvokeCommandEntryPoint from TA 00000000-0000-0000-0000-0053545354ab from firmware version G973FXXU1ASBA (Ghidra screenshot)

In eight other instances we found that the check was omitted altogether. All TAs in question are listed here to facilitate further research, along with Figure 12, which shows the duration of the vulnerability of each TA. On this one may remark, that all vulnerable TAs were as such from the first firmware image, except for TA 00000000-0000-0000-0000-5345435f4652 which was only vulnerable during 1 month, before getting patched in July 2019.

- 00000000-0000-0000-0000-000000000046

- 00000000-0000-0000-0000-000048444350
- 00000000-0000-0000-0000-0000534b504d
- 00000000-0000-0000-0000-00575644524d
- 00000000-0000-0000-0000-42494f535542
- 00000000-0000-0000-0000-46494e474502
- 00000000-0000-0000-0000-53454d655345
- 00000000-0000-0000-0000-5345435f4652 (only for SW REV 2)
- 00000000-0000-0000-0000-0053545354ab
- 00000000-0000-0000-0000-5354494d4552

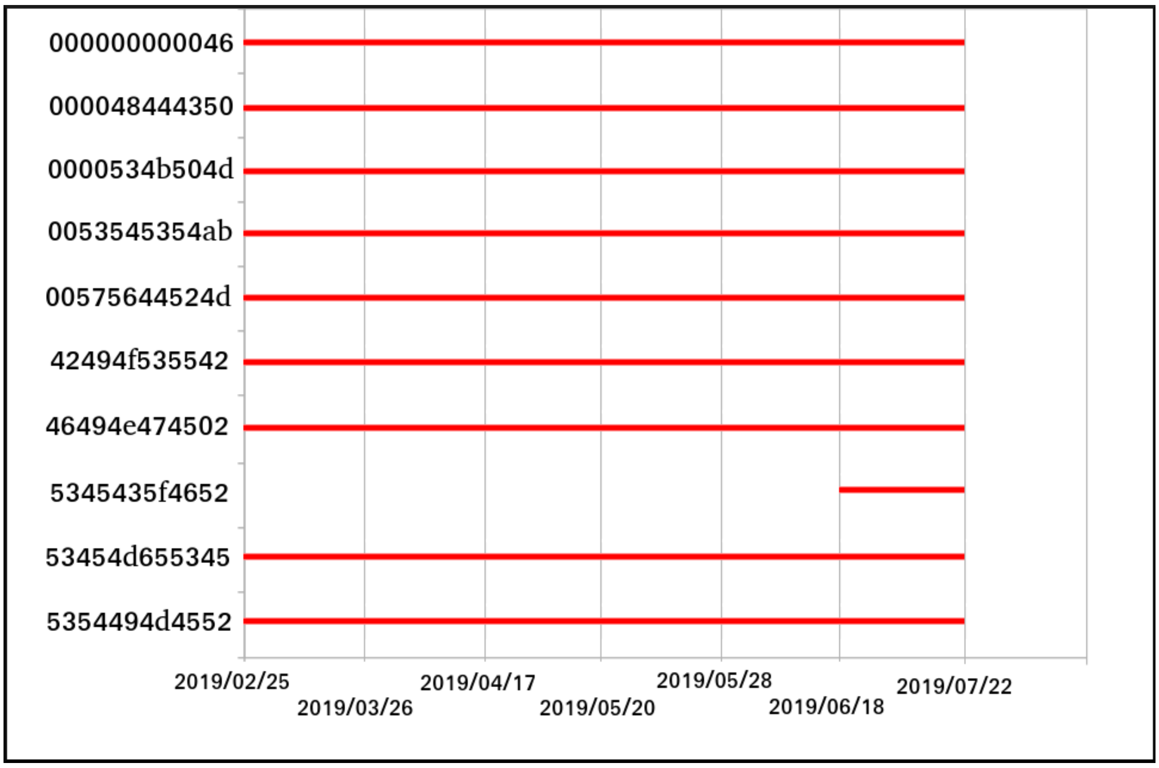


Figure 12: Timeline of vulnerabilities

For the first 7 firmware images we thus have 9-10 vulnerable TAs out of 35, which amounts to 25.71%-28.57%. This shows the seriousness of the type-confusion bug, and indicates that further research is necessary on this topic.

From firmware image G973FXXU3ASG8, which has SW REV code 3, all of the vulnerable TAs were patched. This means that between 2019/06/18 (building date of the previous version) and

2019/07/22, Samsung became aware of this bug, and it was patched right away. We do not know if this was done simply by alerting the developers, who thereafter checked themselvesfrom for the presence of this bug, or if Samsung implemented an automatic detector. The developers have the advantage of having access to the source code, which makes such an analysis easier since they do not have to reverse engineer the TAs with tools such as Ghidra or IDA. But seeing the challenges that we have faced during this research, our suspicion is that the checks are implemented by humans.

## Discussion

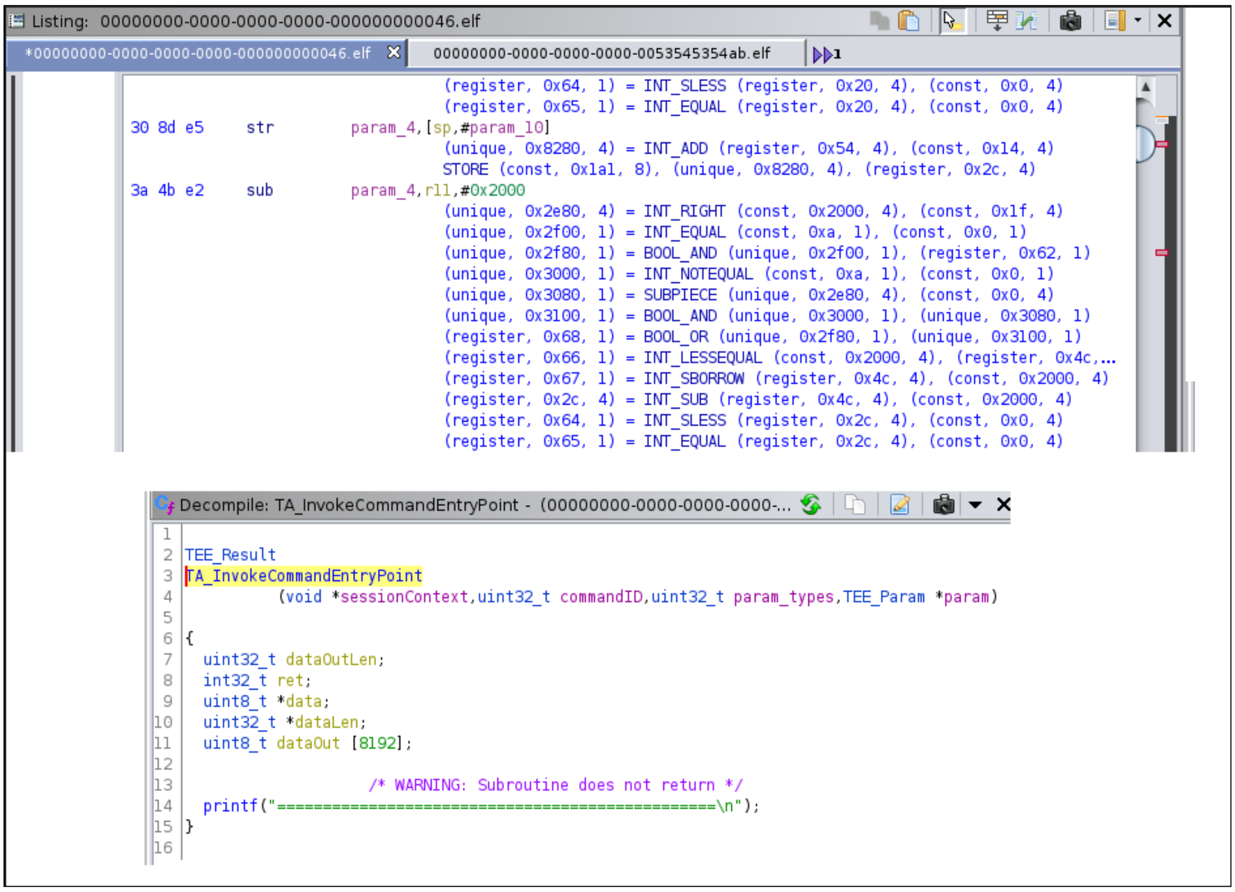
The static analyzer that we further developed during this research functions by analyzing PCODE output from Ghidra. The disassembly and decompilation done by Ghidra is not 100% deterministic, which in turn leads to the static analyzer not being completely reliable. It is for this reason that we did both automatic and manual research. This hybrid method permitted us to verify that no false positives were flagged as vulnerable.

There are two cases in particular which highlight the fallibility of Ghidra. In the first instance (Figure 13), the static analyzer signals a vulnerability, even though in the decompiled C code we see no problem. However, when we read through the assembly code, we can indeed conclude that the parameters are accessed without a previous check. In the second case (Figure 8), Ghidra signals that 16 parameters are given to the function, even though the decompiled C code and manual analysis of the assembly code leads us to believe that there are, in fact, none.

The static analyzer itself has limitations. The TA shown in Figure 14 is flagged as vulnerable, since the parameters are accessed without a previous check. After manual analysis we discovered that the parameters are used in a comparison, which in this case does not constitute harmful behavior. One may then consider changing the static analyzer, to disregard comparisons done on the parameters. This could lead to the compared value being misused, which then would not be signaled. Thus, such a modification would go against our policy of eliminating false negatives.

We use PCODE output in the static analyzer, since this is uniform across all platforms. If one were to build a static analyzer that works on assembly (in our case ARM assembly), this would impact usability on a large scale. However, should one wish to analyze a large number of TAs which all share the same platform, this may be an option to achieve more accurate results.

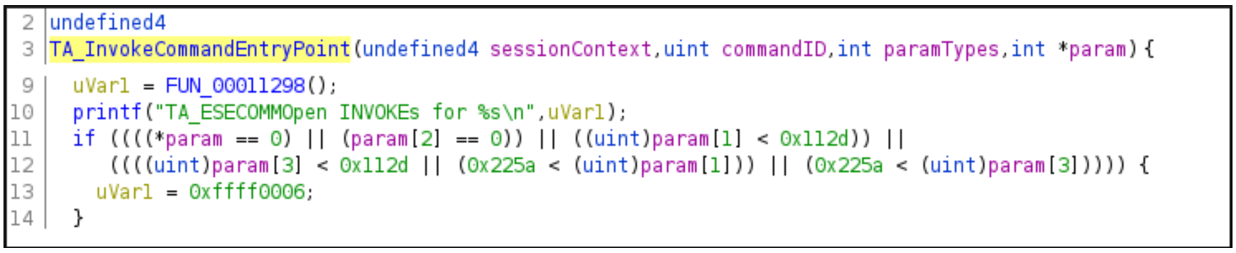
The high-level C code output by Ghidra has been shown to be unreliable in some instances. However, it could be used to complement work done by the static analyzer, since the C code, when correct, shows the parameters used, which would solve the problem of these getting displaced or renamed.


Listing: 00000000-0000-0000-0000-000000000046.elf
*00000000-0000-0000-0000-000000000046.elf
00000000-0000-0000-0000-0053545354ab.elf
Decompile: TA_InvokeCommandEntryPoint - (00000000-0000-0000-0000-...

```
                                     (register, 0x64, 1) = INT_SLESS (register, 0x20, 4), (const, 0x0, 4)
                                     (register, 0x65, 1) = INT_EQUAL (register, 0x20, 4), (const, 0x0, 4)
30 8d e5     str        param_4,[sp,#param_10]
                                     (unique, 0x8280, 4) = INT_ADD (register, 0x54, 4), (const, 0x14, 4)
                                     STORE (const, 0x1a1, 8), (unique, 0x8280, 4), (register, 0x2c, 4)
3a 4b e2     sub        param_4,r11,#0x2000
                                     (unique, 0x2e80, 4) = INT_RIGHT (const, 0x2000, 4), (const, 0x1f, 4)
                                     (unique, 0x2f00, 1) = INT_EQUAL (const, 0xa, 1), (const, 0x0, 1)
                                     (unique, 0x2f80, 1) = BOOL_AND (unique, 0x2f00, 1), (register, 0x62, 1)
                                     (unique, 0x3000, 1) = INT_NOTEQUAL (const, 0xa, 1), (const, 0x0, 1)
                                     (unique, 0x3080, 1) = SUBPIECE (unique, 0x2e80, 4), (const, 0x0, 4)
                                     (unique, 0x3100, 1) = BOOL_AND (unique, 0x3000, 1), (unique, 0x3080, 1)
                                     (register, 0x68, 1) = BOOL_OR (unique, 0x2f80, 1), (unique, 0x3100, 1)
                                     (register, 0x66, 1) = INT_LESSEQUAL (const, 0x2000, 4), (register, 0x4c,...
                                     (register, 0x67, 1) = INT_SBORROW (register, 0x4c, 4), (const, 0x2000, 4)
                                     (register, 0x2c, 4) = INT_SUB (register, 0x4c, 4), (const, 0x2000, 4)
                                     (register, 0x64, 1) = INT_SLESS (register, 0x2c, 4), (const, 0x0, 4)
                                     (register, 0x65, 1) = INT_EQUAL (register, 0x2c, 4), (const, 0x0, 4)
```

```

TEE_Result
TA_InvokeCommandEntryPoint
          (void *sessionContext,uint32_t commandID,uint32_t param_types,TEE_Param *param)

{
  uint32_t dataOutLen;
  int32_t ret;
  uint8_t *data;
  uint32_t *dataLen;
  uint8_t dataOut [8192];

                    /* WARNING: Subroutine does not return */
  printf("=================================================\n");
}

```

Figure 13: Vulnerable TA_InvokeCommandEntryPoint from TA 00000000-0000-0000-0000-000000000046 from firmware version G973FXXU1ASBA (Ghidra screenshot)



```
undefined4
TA_InvokeCommandEntryPoint(undefined4 sessionContext,uint commandID,int paramTypes,int *param){
  uVar1 = FUN_00011298();
  printf("TA_ESECOMMOpen INVOKEs for %s\n",uVar1);
  if ((((*param == 0) || (param[2] == 0)) || ((uint)param[1] < 0x112d)) ||
     ((((uint)param[3] < 0x112d || (0x225a < (uint)param[1])) || (0x225a < (uint)param[3])))) {
    uVar1 = 0xffff0006;
  }
```

Figure 14: Safe, but flagged TA_InvokeCommandEntryPoint from TA 00000000-0000-0000-0000-657365636f6d from firmware version G973FXXU1ASBA (Ghidra screenshot)

In my opinion, in order to develop the most reliable static analyzer all three approaches should be combined (PCODE, assembly, C).

Because our analysis was done as part of a Bachelor project, this work represents a part of a more extensive project. Further research is necessary, to some extent on different Samsung models, but more importantly, on TAs of other vendors, who may not be aware of this bug, or whohave fixed it at a different time than Samsung.

During our research, we came across TAs compiled with Rust, a memory-safe language. At first glance, one may think that memory safety guaranteed by Rust at compilation could be the antidote to the type-confusion bug. However, using the OPTEE environment and QEMU, we were able to compile an unsafe TA, pass it an arbitrary value and crash the program.

## Related Work

Regading the phenomenon of type-confusion bugs researchers have suspected the existence of them since TEEGRIS and TrustZone-based TEEs in general, were introduced. According to Samsung's website [9], in 2019 the Taiwanese TeamT5 disclosed to them privately the presence of eight type-confusion bugs. They did not, however, publish their findings in a paper, or otherwise make the analysis methods public. According to the CVE database [4], a 9th type-confusion bug was discovered in 2020, and patched by Samsung in April, 2020. Samsung's website does not even mention CVE-2020-11603/SVE-2020-16599 [10], so the documentation of this bug is unclear. We found that the TA in question (MLDAP Trustlet) was patched earlier, and thus we tend to think that the CVE database is incorrect.

In their 2020 paper [3] Cerdeira et al. discuss in a more global manner the security concerns of TrustZone-assisted TEE Systems such as TEEGRIS. They mention that lack of input checks may lead to vulnerabilities, but they do not explicitly mention the type-confusion bugs.

Busch et al. wrote a paper in 2020 [2], which examines Huawei's TrustedCore TEE. This paper is also more globally focused, but contains useful strategies an attacker may use once they have gained control over a TA. They dive into some detail regarding TAs, but they also do not explicitly state the existence of type-confusion bugs in these TAs. As mentioned in the Discussion section, the TAs found in Huawei firmware images would also be a worthwhile research subject.

The article by riscure.com [7, 8], describes and presents an in-depth analysis of the type-confusion bug, with possible attack scenarios. But it does not provide any information as to when the bug appeared, how many TAs are affected, or which firmware images are safe.

Taking a different approach than the aforementioned sources, Harrison et al. wrote a paper [6] on emulating TEEs for dynamic analysis. Among others, they analyzed TEEGRIS, and indeed found the type-confusion bug. As it is not the main focus of their paper, they do not go into details about how many instances of the bug they found, or on which devices.

Finally, in his PHD thesis [1], Marcel Busch describes in detail the GP API, the type-confusion bug in Huawei TAs and the original static analyzer. Even though his dissertation is also of a more global approach than this report, the topic of our research is closely tied to a subtopic of his thesis. But while his focus is on Huawei, ours is on Samsung.

While this report is not comparable in depth to the previously mentioned sources, it provides a timeline of the vulnerable firmware images and a comprehensive analysis of vulnerable TAs in firmware images available for the S10 device.

## Conclusion

The main result of this project is the discovery that on the Samsung S10 device, from 2019/02/25 to 2019/07/22, 9-10 different TAs were vulnerable to the type-confusion bug detailed in this report. This vulnerability can not be considered an accident, asproven by the large percentage of vulnerable TAs (26-29%) and by the fact that even when the type check was carried out, this did not always guarantee memory-safety. Thus, we must come to the conclusion that the problem is the result of a TEE design that did not factor in this particular bug. In light of this, more extensive data research is necessary to accurately assess the extent of this vulnerability.

# Bibliography

[1] Marcel Busch. “On the Security of ARM TrustZone-Based Trusted Execution Environments”. doctoralthesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2021, p. 130.

[2] Marcel Busch, Johannes Westphal, and Tilo Mueller. “Unearthing the TrustedCore: A Critical Review on Huawei’s Trusted Execution Environment”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: `https://www.usenix.org/conference/woot20/presentation/busch`.

[3] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1416–1432. DOI: `10.1109/SP40000.2020.00061`.

[4] cve.mitre.org. *Vulnerability Database*. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11603`. 2020.

[5] globalplatform.org. *GlobalPlatform Technology TEE Internal Core API Specification Version 1.1.2.50 (Target v1.2)*. `https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf`. 2018.

[6] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. “PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation”. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, 2020, pp. 789–806. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/harrison`.

[7] riscure.com. *Breaking TEE Security Part 1: TEEs, TrustZone and TEEGRIS*. `https://www.riscure.com/blog/tee-security-samsung-teegris-part-1`. 2021.

[8] riscure.com. *Breaking TEE Security Part 2: Exploiting Trusted Applications (TAs)*. `https://www.riscure.com/blog/tee-security-samsung-teegris-part-2`. 2021.

[9] security.samsungmobile.com. *Security Updates*. `https://security.samsungmobile.com/securityUpdate.smsb`, 2019 August.

[10] security.samsungmobile.com. *Security Updates*. `https://security.samsungmobile.com/securityUpdate.smsb`, 2020 April.