GlobalConfusion: TrustZone Trusted Application 0-Days by Design

Marcel Busch, Philipp Mao, and Mathias Payer, *EPFL*

https://www.usenix.org/conference/usenixsecurity24/presentation/busch-globalconfusion

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

GlobalConfusion: TrustZone Trusted Application 0-Days by Design

Marcel Busch Philipp Mao Mathias Payer
EPFL, Lausanne, Switzerland

Abstract

Trusted Execution Environments form the backbone of mobile device security architectures. The *GlobalPlatform Internal Core API* is the de-facto standard that unites the fragmented landscape of real-world implementations, providing compatibility between different TEEs.

Unfortunately, our research reveals that this API standard is prone to a design weakness. Manifestations of this weakness result in critical type-confusion bugs in real-world user-space applications of the TEE, called Trusted Applications (TAs). At its core, the design weakness consists of a fail-open design leaving an optional type check for untrusted data to TA developers. The API does not mandate this easily forgettable check that in most cases results in arbitrary read-and-write exploitation primitives. To detect instances of these type-confusion bugs, we design and implement GPCheck, a static binary analysis system capable of vetting real-world TAs. We employ GPCheck to analyze 14,777 TAs deployed on widely used TEEs to investigate the prevalence of the issue. We reconfirm known bugs that fit this pattern and discover unknown instances of the issue in the wild. In total, we confirmed 9 known bugs, found 10 instances of silently-fixed bugs, and discovered a surprising amount of 14 critical 0-day vulnerabilities using our GPCheck prototype. Our findings affect mobile devices currently in use by billions of users. We responsibly disclosed these findings, already received 12,000 USD as bug bounty, and were assigned four CVEs. Ten of our 14 critical 0-day vulnerabilities are still in the responsible disclosure process. Finally, we propose an extension to the GP Internal Core API specification to enforce a fail-safe mechanism that removes the underlying design weakness. We implement and successfully demonstrate our mitigation on OPTEE, an open-source TEE implementation. We shared our findings with GlobalPlatform and suggested our mitigation as an extension to their specification to secure future TEE implementations.

1 Introduction

The security architecture of modern mobile devices heavily relies on *Trusted Execution Environments (TEEs)* to protect highly sensitive data for use cases such as biometric authentication, secure storage, digital rights management, and mobile payment. These distinct use cases require modularization and isolation. They are therefore typically encapsulated in user-space programs executed within the TEE called *Trusted Applications (TAs)*.

The organic evolution of the mobile ecosystem led to heavy fragmentation and heterogeneity of the software stacks on mobile devices. Depending on OEM (Original Equipment Manufacturer), chipset, and model, several TEEs, including QSEE, Kinibi, TEEGris, Trusty, OPTEE, and BeanPod power the security backend of our devices. Each TEE exposes its unique API that is incompatible with the other TEEs.

To counteract this fragmentation and ensure the compatibility of TAs across different TEE implementations, a non-profit industry association, called GlobalPlatform strives to enable a collaborative and open ecosystem by developing specifications, especially regarding trusted computing technologies and particularly for TEEs. The specification relevant to our work is the GP TEE Internal Core API [22], which defines a common interface that can be used by TAs.

In this work, we take a closer look at the GP TEE Internal Core API design and make an insightful discovery. The design choices of this de-facto standard API led to a class of type-confusion bugs affecting almost all TEE implementations used on production devices. Our analysis reveals that this design weakness affected more TAs than was publicly known and continues to manifest itself as critical vulnerabilities in TAs deployed on mobile phones used by billions of users.

In detail, the design weakness leads to a type-confusion bug where an attacker-controlled value is used as a memory reference. In the majority of these cases, this bug leads to arbitrary read-and-write exploitation primitives in the context of the affected TA and, thus, allows to fully take control over parts of the TEE.

Prior work noticed instances of these bugs in isolation without identifying the design weakness. We are the first to elaborate on the design weakness of the GP Internal Core API and its ecosystem-wide scale. Our discovery connects seemingly unrelated bugs affecting platforms of reputable vendors like Huawei [3, 54] and Samsung [43, 49], and relates these disclosures back to the GP API design weakness.

We present several findings and contributions. First, we detail and systematize the design weakness in the GP TEE Internal Core API highlighting the severe class of type-confusion bugs this weakness leads to. Next, we aim to study the prevalence of this bug class in the ecosystem. Since the vast majority of real-world TAs are closed-source, we designed and implemented GPCheck, a static binary analysis system that automatically detects instances of the type-confusion bug. In the design of GPCheck, we model this domain-specific class of type-confusion bugs as taint-style vulnerability. The encoding of this vulnerability class is our novel contribution, allowing us to conduct a large-scale study regarding the prevalence of this bug class in practice. In total, we investigate 14,777 TAs deployed on billions of devices by 5 vendors. Our results reconfirm the 9 previously known instances of the bug class. Unfortunately, the scope and impact of this bug class is larger than publicly known. We uncover 10 *unknown* and silently fixed vulnerabilities in old TAs, and 14 critical *0-day* vulnerabilities in the latest versions of TAs. Further, to mitigate the threat of this design weakness and to end this stream of critical type-confusion bugs, we propose an extension to the GP TEE Internal Core API specification. This extension preserves compatibility and adds a fail-safe design feature that will prevent type-confusion bugs in the future. We implemented this design on OPTEE and demonstrate its effectiveness.

In summary, we make the following contributions:

- Discovery of a design weakness in TAs using the GlobalPlatform API specification that leads to a series of critical bugs across vendors.
- Modeling of this TA type-confusion bug class as a taint-style vulnerability, as well as the design and implementation of a static analysis system, GPCheck, capable of detecting instances of this bug class in closed-source TAs.
- Automated static analysis pipeline to conduct a large-scale study to measure the scope and impact of this threat. We analyze 14,777 TAs spanning 5 vendors and demonstrate the scope of this issue that affects the majority of the ecosystem.
- Proposal and implementation of a viable countermeasure based on OPTEE.

The artifacts of our research are publicly available at https://github.com/HexHive/GlobalConfusion. All discovered bugs were responsibly disclosed to the respective vendors. We are in an ongoing responsible disclosure for 10 vulnerabilities, 4 CVEs have been assigned. Additionally, we contacted GlobalPlatform aiming to change the underlying specification to mitigate this bug class once and for all.

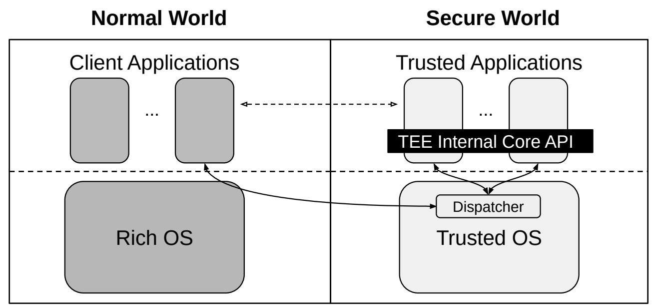

Figure 1: Architecture and communication channels of modern TrustZone-based TEEs. The logical channel of a Client Application–TA interaction (dashed line) is carried out by both OSes that, cooperatively, forward and dispatch requests (solid lines).

2 Background

The basis for almost all TEEs found on modern mobile devices is ARM TrustZone [6]. It allows for partitioning of the System-on-Chip (SoC) into two execution contexts – the Secure World and the Normal World – where code and data from the Secure World cannot be accessed by the Normal World. The idea is to run a feature-rich operating system (the rich OS) and its userland in the Normal World and only execute trusted code in the Secure World. Recent TrutZone-based TEEs split the Secure World into a kernel, the trusted OS, and a userland, which hosts TAs.

For data exchange, a logical communication channel exists between a Client Application and a TA (dashed line in Figure 1). Using this channel, Client Applications can request services from TAs. For example, requesting the generation of an asymmetric key pair, where the private key resides in the TEE, and the Client Application can use the public key [23]. In addition to the key generation, the TA would also provide an API to perform cryptographic operations using the safely stored private key (*e.g.,* sign or decrypt messages).

Technically, the Client Application cannot call a TA directly. It must go through the rich OS that takes care of using a Normal World-Secure World shared memory region for the provided request data and initiates the world switch using a privileged instruction (*e.g.,* `smc`). Then, the trusted OS dispatches the request to the TA. When the TA has processed the request, it writes its output to the shared memory region used for this session, and returns to the trusted OS, which, in turn, initiates the world switch back to the Normal World. Finally, the rich OS returns execution to the Client Application. Figure 1 depicts this communication channel with solid lines.

Vendors of TEEs have an interest in providing a common interface for TAs in order to execute third-party TAs on their platforms. One set of standards that has been adopted by the mobile market is specified by GlobalPlatform (GP). GP is a non-profit organization dedicated to fostering open ecosystems through the development of specifications, with a focus on TEEs. The GP TEE Internal Core API [22] defines a common interface that can be used by TAs (depicted in Figure 1). This API is the de-facto standard for developing TAs that may be deployed on multiple different TEE implementations.

One especially relevant part of the GP TEE Internal Core API is the lifecycle functions that are used from a Client Application's perspective to interact with a TA. The specification defines the following lifecycle functions:

1. `TA_CreateEntryPoint`: This function is the constructor for a TA. It is called only once during the lifetime of the TA when the first session is established.
2. `TA_OpenSessionEntryPoint`: This function is used to establish a session between the Client Application and the TA. It is primarily responsible for authenticating the Client Application and initializating data structures for the session.
3. `TA_InvokeCommandEntryPoint`: This function invokes the actual command handler of the TA. Each TA usually implements multiple commands.
4. `TA_CloseSessionEntryPoint`: This function frees all session-specific state.
5. `TA_DestroyEntryPoint`: This function deallocates all resources reserved in the initial entry point creation.

Both the `TA_OpenSessionEntryPoint` and `TA_InvokeCommandEntryPoint` handle untrusted data passed as arguments. In Section 4, we provide an example of how their *fail-open* design leads to critical type-confusion vulnerabilities.

3 Threat Model

We adopt our threat model based on the standard assurances provided for TrustZone-based Trusted Execution Environments (TEEs), tailoring it to the specific environment of a functional mobile device operating on the Android platform. TEEs leveraging TrustZone offer robust hardware-based isolation, ensuring the integrity and confidentiality of all components within the Secure World. This isolation thwarts the execution of unauthorized code within the TEE, such as loading unsigned or modifying existing code. It also safeguards confidential information, including cryptographic keys and biometric identifiers, from being exposed to the Normal World.

In this threat model, we assume an adversary with the capability to execute code in the context of the rich OS and Client Applications, with access to TEE-exposed interfaces.

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

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

Listing 1: The function signatures of the two affected GP API functions.

This attacker model aligns with real-world scenarios on typical Android devices, considering the historical prevalence of privilege escalation exploits targeting the Linux kernel, exemplified by incidents like CVE-2018-9568, CVE-2019-2215, and CVE-2022-0847. Furthermore, the model can be nuanced and weakened, given that a plethora of user-space daemons have access to the TEE driver interface exposed by the kernel. In many cases, access to this interface (limited to individual Client Applications) is sufficient for an attacker to execute the described attacks.

Our specific focus involves an adversary aiming to compromise the integrity and confidentiality of a Trusted Application (TA) and potentially exploit the TA's capabilities to breach the TEE's operating system. Note that achieving the latter only requires an exploitable vulnerability in a single TA, establishing a single point of failure.

4 GlobalConfusion: A Design Weakness in the De-Facto TA API Standard

We discovered a design weakness related to the GlobalPlatform API specification for TAs that proposes a fail-open design for optional, but critical, type checks of untrusted parameters. In the majority of cases, manifestations of this weakness result in an arbitrary read-and-write exploitation primitive and, thus, a complete compromise of the affected TA.

The two affected GP API function signatures are illustrated in Listing 1. In all GP-compliant TAs, these functions are directly exposed to Client Applications and the rich OS which makes them accessible in the context of our threat model described in Section 3. Note that untrusted input is crossing the trust barrier between Normal World and Secure World when these functions are invoked and, thus, all arguments must be properly sanitized.

The `sessCtx` is an opaque pointer that can be set by the TA during session establishment and does not contain untrusted data. Within subsequent command invocations, this pointer is available and used to maintain session-specific data structures. The remaining parameters contain untrusted data and their usage becomes clear by looking at an example.

In Listing 2, we present an echo-TA to demonstrate the usage of GP-compliant TAs and to highlight the type-confusion

```c
1  TEE_Result TA_InvokeCommandEntryPoint(
2      void __maybe_unused *sessCtx,
3      uint32_t cmdId, uint32_t paramTypes,
4      TEE_Param params[4])
5  {
6    (void)&sessCtx; // Unused parameter
7
8    switch (cmdId) {
9    case TA_ECHO_CMD_ECHO: {
10     char *in_buf, *out_buf;
11     size_t in_buf_sz, out_buf_sz;
12
13     /*
14     uint32_t exp_paramTypes = TEE_PARAM_TYPES(
15                     TEE_PARAM_TYPE_MEMREF_INPUT,
16                     TEE_PARAM_TYPE_MEMREF_OUTPUT,
17                     TEE_PARAM_TYPE_NONE,
18                     TEE_PARAM_TYPE_NONE);
19
20     if (paramTypes != exp_paramTypes)
21       return TEE_ERROR_BAD_PARAMETERS;
22     */
23
24     in_buf_sz = params[0].memref.size;
25     in_buf = (char*)params[0].memref.buffer;
26     out_buf_sz = params[1].memref.size;
27     out_buf = (char*)params[1].memref.buffer;
28
29     if (in_buf_sz > out_buf_sz)
30       return TEE_ERROR_BAD_PARAMETERS;
31
32     TEE_MemMove((void*)out_buf, in_buf, in_buf_sz);
33
34     return TEE_SUCCESS;
35   }
36   default:
37     return TEE_ERROR_BAD_PARAMETERS;
38   }
39 }
```

Listing 2: The GP TEE Internal Core API specification proposes a weak "fail-open" design to sanitize untrusted input. The type check that distinguishes `memref` from `value` parameters is optional.

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

```c
#define TEE_PARAM_TYPE_NONE          0
#define TEE_PARAM_TYPE_VALUE_INPUT   1
#define TEE_PARAM_TYPE_VALUE_OUTPUT  2
#define TEE_PARAM_TYPE_VALUE_INOUT   3
#define TEE_PARAM_TYPE_MEMREF_INPUT  5
#define TEE_PARAM_TYPE_MEMREF_OUTPUT 6
#define TEE_PARAM_TYPE_MEMREF_INOUT  7

#define TEE_PARAM_TYPE_GET(t, i) \
    ((((uint32_t)t) >> ((i)*4)) & 0xF)
```

Listing 3: The parameter type is either a `value` or `memref` struct. Due to the `union`, the two structs overlap in memory. The GP API specifies two classes of parameter types, `value` and `memref`. Each parameter's type is encoded by 4 bits in the `param_types` argument.

problem exposed through the API. In this listing, we can see a simple command handler of an echo-TA. The session context (`sessCtx` in Lines 2 and 6) is not used in this example, and the command handler only implements one command identifier (`TA_ECHO_CMD_ECHO` in Line 9). The implemented echo command uses the first parameter (`params[0]` in Lines 24 and 25) as a reference to an input buffer and the second parameter (`params[1]` in Lines 26 and 27) as a reference to an output buffer. As the name of this TA suggests, the input buffer is copied to the output buffer (Line 32) in this command handler before the TA returns successfully.

Each GP-compliant TA must check the parameter types (`paramTypes`) sent by a client against the expected parameter types (Lines 14 to 21) for the corresponding `cmdId` handler. If the expected types do not match the provided types, the TA is supposed to abort execution and return a `TEE_ERROR_BAD_PARAMETERS` status code to the client. The individual parameters (`params`) are supposed to be passed using either one of two different parameter type classes, `value` or `memref`. Each class can be an input, output, or in-out parameter (Listing 3). Additionally, there is the `none` type indicating that a parameter is not used. Each parameter is implemented as a `union`, `TEE_Param`, consisting of a `memref` and a `value` struct, as illustrated in Listing 3. The `memref`'s first member is an opaque pointer to a buffer, and its second member describes the size of this buffer. The `value`'s members, `a` and `b`, are two 32-bit unsigned integers. Due to the `union`, the two structs overlap, and the same underlying data can be interpreted as either `memref` or `value` type. In a 64-bit environment, the integers `a` and `b` overlap with only the `buffer`.

Design Weakness. In Listing 2, we indicate a forgotten parameter type check in Lines 14 to 21. This type check is essential to prevent type-confusion vulnerabilities. The GlobalPlatform TEE Internal Core API specification proposes an *optional* preprocessor macro-based type check, resulting in a *weak fail-open design*, instead of enforcing the proper sanitization of untrusted input (fail-closed design). Listing 2 demonstrates a manifestation of this weak design when a `value` is used like a `memref`. Assuming an adversary notices the missing type check in any of the commands of any TA (usually there are many commands), it can have severe consequences as we can learn from our example echo-TA. In this case, an attacker can invoke the `TA_ECHO_CMD_ECHO` command and provide the type information used for any of the two used parameters to indicate a `value` type. Since the TA does not check its parameter types, but still interprets them as `memref` types, the attacker provided `a` and `b` values overlap with the `buffer` and `size` fields (in the 32-bit case) and, thus, the attacker has control over the two pointers within the virtual address space of the TA that can be read from or written to. While the prior case can lead to the leakage of confidential data, such as reading export-protected cryptographic keys, the latter case could allow an adversary to manipulate the TA's address space and, in the worst case, lead to code execution.

5 GPCheck Design

Based on the design weakness of the GlobalPlatform Internal Core API as introduced in Section 4, we first surveyed the landscape of existing analysis tools and second, due to their limitations, design a static binary analysis tool to automatically find manifestations of this weakness in proprietary closed-source TAs.

5.1 Prior Art

In our study, we aim for a large-scale analysis of closed-source TAs compiled from C/C++ code. These TAs are shipped as binary blobs in proprietary firmware images and deployed on production devices. They are executed within the TEE of these production devices and do not allow any introspection. Within these TAs, we aim to detect a type-confusion vulnerability as outlined in Section 4. This section reviews prior work related to this class of vulnerabilities. An overview of prior work is summarized in Table 1.

Dynamic Approaches. At its core, manifestations of the GlobalPlatform Internal Core API design weakness materialize as type-confusion vulnerabilities in TAs. Prior research on type-confusion vulnerability detection focuses primarily on dynamic analysis techniques which fundamentally requires concrete executions to trigger the type-confusion. TypeSan [25], HexType [28], BiType [31], EffectiveSan [16], and Uncontained [32] propose several approaches in different domains to instrument target programs during compilation. This instrumentation allows for the detection of type confusions during runtime. In principle, dynamic taint tracking approaches like TaintDroid [18] or TaintArt [56] could also be leveraged to uncover type-confusion vulnerabilities by ensuring that a type check occurs before every access to the variably typed object. There exist similar dynamic approaches targeting type confusions in binaries, including libcrunch [30] and BinTyper [31]. Unfortunately, all of these approaches require recompilation based on source code to extract the necessary type information, require the modification of the target program to add the checks, and/or rely on powerful introspection capabilities into the program state. TAs are closed-source binaries compiled from C/C++. They are vendor-signed and cannot be executed when modified. Additionally, TrustZone TEEs enforce isolation between the Normal World and the TEE, meaning that introspection is prohibited. While emulation-based approaches for TEEs [26] seem promising, they suffer from fidelity issues, require significant engineering efforts to support a wider range of TEE implementations, and existing prototypes are not available to the public. Hence, past approaches leveraging dynamic analysis are inappropriate to support our study of type-confusion bugs in TAs.

Static Approaches. As illustrated in Section 4, the type-confusion bug becomes a vulnerability when a `memref` use is not preceeded by a corresponding type check. In static analysis, this problem falls into the category of *taint-style vulnerabilities*. Analyses aiming to detect taint-style vulnerabilities consist of (i) taint-introducing sources, (ii) taint propagation rules, (iii) taint-removing sanitizers, and (iv) taint-consuming sinks. These analyses raise alerts when a taint can flow from source to sink without being sanitized.

The encoding of these four elements of taint-style vulnerability analyses is highly problem-specific. For instance, Livshits and Lam [38] define a taint-style vulnerability analysis for Java-based web applications. Amongst other encodings, they detect SQL injections when tainted data from an untrusted source flows into an SQL statement interpreted by the database management system without being sanitized. Similarly, Pixy [29] encodes a taint-style vulnerability addressing cross-site scripting vulnerabilities in PHP applications. Their analysis raises an alert when untrusted input is returned to the client without being sanitized.

Different from these domains, our targets are binaries compiled from C/C++. Taint-style vulnerability analyses for binaries were used by Bootstomp [46] and its successor Karonte [47]. Both systems use heuristics, i.e., keywords in strings and `memcpy` functions, to identify sources and sinks. In the scenarios for these systems, sanitizers are not well-defined and it is left to a human analyst to decide if a detected flow leads to unintended behavior. COMfusion [65] is another system leveraging taint-style vulnerability analyses to detect improper usages of unions in the context of Microsoft Component Object Model (COM) code. COMfusion requires Microsoft Interface Definition Language (MIDL) files and uses them to identify union-type parameters of functions. These parameters serve as sources for the inter-procedural taint analysis. The taints are propagated using no further specified "normal data movements". Interestingly, the sinks are identified by a succeeding symbolic execution phase as specific uses of tainted data (i.e., a union is passed as a parameter to `memcpy`). The sanitization check is integrated to the symbolic execution by introducing the type selector (i.e., member of the `struct` containing the union) as symbolic variable and determining its possible values at each sink location. If there are more than one possible values (i.e., multiple types possible), the system reports a type confusion.

The above-mentioned systems propose promising approaches to detect domain-specific taint-style vulnerabilities. However, they are not directly applicable to the type-confusion issues outlined in Section 4. Bootstomp and Karonte do not rely on sanitizers and leave the interpretation of alerts to human analysts. Given the large-scale scope of our study, this approach is infeasible. Further, while COMfusion introduces a taint for one union-typed parameter, our problem (see Section 4) requires tracing the `params` array and continue as a multi-tag taint system whenever the `TEE_Param` union members of this array are propagated. To be precise, we are only interested in the propagation of the first member

Solution	Analysis	Target	Approach
TypeSan [25]	Dynamic	Source (C++)	Runtime Sanitization
HexType [28]	Dynamic	Source (C++)	Runtime Sanitization
BiType [44]	Dynamic	Source (C++)	Runtime Sanitization
Effectivesan [16]	Dynamic	Source (C++)	Runtime Sanitization
Uncontained [32]	Hybrid	Source (C)	Def-Use Chains / Runtime Sanitization
TaintDroid [18]	Dynamic	Bytecode (Dalvik)	Runtime Taint Tracking
TaintArt [56]	Dynamic	Bytecode (Dalvik)	Runtime Taint Tracking
libcrunch [30]	Dynamic	Binary (C++)	Runtime Sanitization
BinTyper [31]	Hybrid	Binary (C++)	Class Recovery / Runtime Sanitization
Bootstomp [46]	Static	Binary (C)	Static Taint Propagation
Karonte [47]	Static	Binary (C)	Static Taint Propagation
COMfusion [65]	Static	Binary (C)	Static Taint Propagation

Table 1: Overview of prior approaches related to type-confusion detection.

(buffer or value) of TEE_Param. COMfusion does not support multiple taints and cannot express these tainting rules with sufficient granularity. Extending COMfusion in these two aspects would be an option, but neither the prototype source code nor any other artifacts of this work are publicly available. In the following sections, we describe our contribution of encoding the type-confusion bug class into a taint-style vulnerability and our system, GPCheck, capable of detecting instances of these vulnerabilities in closed-source TAs.

5.2 Overview

Our system, GPCheck, relies on static analysis to find the type-confusion vulnerabilities as mentioned above. We extend commodity reverse-engineering tools to implement our static analyses. In particular, we require a tool that disassembles the TA's machine code and identifies functions. Further, we aim for an architecture-agnostic analysis by utilizing an intermediate representation (IR) typically provided by reverse engineering tools as a lifter step in the decompilation process. A common feature of these tools is to optimize the IR representation of the code in multiple passes. These optimizations, similar to compiler optimizations, transform the IR into the single static assignment (SSA) form to allow for more advanced data-flow analyses. For instance, SSA facilitates the detection of def-use chains of variables because each variable is defined exactly once. All major reverse-engineering tools provide a scriptable API to query their optimized IR. For instance, IDA Pro lifts machine code to Microcode IR [24], Binary Ninja to BNIL [1], and Ghidra to PCODE [2].

Combining powerful and mature decompilation tools with security-centered static analysis is a promising approach to analyzing proprietary TAs as found within firmware images deployed on production devices.

As illustrated in Section 4, the type-confusion bug becomes a vulnerability when memref TEE_Params are used without the corresponding preceding paramTypes check. The core of our system is an information flow analysis that tracks the flow of data within the program. Using this analysis, we can effectively determine the distinct cases of unchecked memref

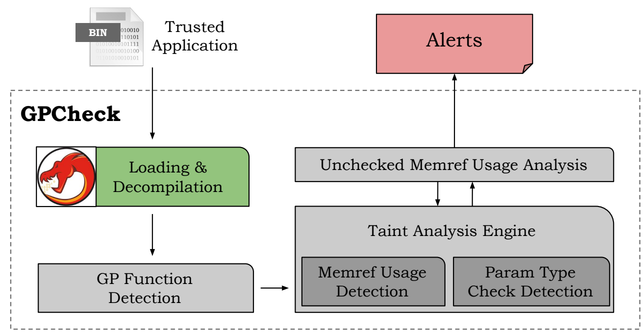

Figure 3: GPCheck consumes closed-source TAs and is based on commodity reverse-engineering tools. First, it detects the relevant GP functions, next it introduces and tracks several taints, and finally combines these information flow analyses to generate alerts for unchecked memref usages.

usages and raise an alert for each of these cases. A human analyst can then process these alerts and verify if the reported type-confusion bugs can be exploited.

Our system proceeds in the following steps, as outlined in Figure 3:

1) TA Pre-processing. GPCheck's input is a TA binary. The design choice of supporting binaries enables our system to handle proprietary closed-source TAs. These binaries can directly be obtained from firmware images of devices employing TEEs. We leverage commodity reverse-engineering tools and their decompilation features to obtain control-flow and data-flow information.

2) GP Function Detection. TA binaries are often stripped. Thus, GPCheck needs to identify the two lifecycle entrypoints TA_InvokeCommandEntryPoint and TA_OpenSessionEntryPoint. Since each TEE implementation uses a different SDK to implement TAs, this detection is TEE-specific and non-trivial. GPCheck identifies these lifecycle entrypoints using a set of effective heuristics that rely on structural patterns of the individual SDKs. The localization of these entrypoints is a prerequisite for our information flow analysis.

3) Memref Usage and Type Check Detection. As explained in Section 4, we are interested in all cases where one of the four TEE_Params may be used as a memref. In particular, any access (dereference) of the buffer member of memref can be critical. Hence, GPCheck considers the two relevant lifecycle functions as taint sources and marks their params argument as tainted. Further, the system tracks the usages of params throughout the CFG and introduces new distinct taint sources for all sinks that access the memory backing any of the up-to-four memref.buffer members. In other words, we convert the params taint into a multi-tag taint with different propagation rules whenever a potential memref.buffer members is created. These memref.buffer candidate taints are further propagated and dereferencing accesses are marked as sinks. Analogously, GPCheck tracks the usages of `paramTypes` and marks all sinks that compare this value. This comparison indicates a parameter type check.

If any taint is propagated to another function, GPCheck properly forwards this information and recursively tracks taints across function boundaries, making this analysis inter-procedural.

4) Unchecked Reachability Analysis and Alerting. Having identified `memref` usages and `paramTypes` checks inter-procedurally, GPCheck performs a reachability analysis where we determine if a `memref` usage sink is reachable from a source (i.e., entrypoint of a function) without encountering a sanitizing `paramTypes` check in the CFG. In other words, "can we find a path within the CFG that connects a source with a sink without traversing a checker node". GPCheck raises an alert for all unchecked usages and compiles a report for a human analyst containing relevant context information to pinpoint the vulnerability during a manual analysis.

The unchecked reachability analysis also decides if descending into tainted callees is necessary. If the call site is checked, GPCheck does not descend into the callee.

False Positives. Conservative static information flow analyses as the one suggested above have high false-positive rates due to overtainting. GPCheck employs a domain-specific analysis tailored for a fairly narrow class of bugs. Due to the introduction of taints via standardized APIs, we assume a homogeneous handling of the tainted parameters. For instance, it is unusual to encounter `paramTypes` checks more than two callees deep into the callgraph originating from the GP API function. Additionally, we observed a uniform pattern to access the `buffer` members inside of the `params` and did not observe any obfuscation or anti-analysis techniques in in-the-wild TAs. These properties are beneficial for static information flow analysis and result in a false-postive rate of about 10%, as our empirical evaluation in Section 7 shows.

5.3 GP Function Detection

In this section, we describe how we can identify GP API functions in proprietary and stripped binaries.

The GP API defines return codes to indicate certain error conditions. For instance, `0xFFFF000C` indicates an out-of-memory condition, while `0xFFFF0000` indicates a generic error. As a heuristic, GPCheck uses these return codes to identify GP function candidates. Additionally, we use a set of vendor-specific constants and (log) strings that consistently appear within TAs, making use of GP "artifacts" to extend the set of candidates.

Next, we use a set of vendor-specific structural features to first split the result set into `TA_OpenSessionEntryPoint` and `TA_InvokeCommandEntryPoint` candidates, and second uniquely identify the two functions. One structural feature is related to the integration of the GP API lifecycle functions into the binary. All lifecycle functions are only referenced once from one function that implements the lifecycle state machine. Therefore, we filter the result set for candidates having only one caller. Then, we exploit the fact that the lifecycle functions have a common caller and cluster our result set into groups of functions having the same caller. A further cross-vendor structural feature is the parameters used by the two target functions. By considering the parameter count of each function, we can exclude candidates and split each group into `TA_OpenSessionEntryPoint` and `TA_InvokeCommandEntryPoint` candidates. Lastly, we end up with groups of candidates for both functions and use the fact that their common caller passes the same variables (`paramTypes` and `params`) to both functions.

In practice, these heuristics yield reliable results. Depending on the target, the used heuristics can be extended or modified to fit the structural features of other implementations.

5.4 Static Information Flow Analysis

After identifying the two problematic functions as described above and using commodity reverse-engineering tools to obtain control- and data-flow information, the core of our static analysis consists of three elements. First, we identify the CFG nodes that check the parameter types (type check). Second, we identify the CFG nodes accessing the memory backing *memref* buffers (memref usage). Third, we determine if any of the *memref* usages is reachable without traversing a type check.

Tracking `paramTypes`. To intra-procedurally track the `paramTypes`, we taint the corresponding parameter of the taint-introducing function (i.e., `TA_OpenSessionEntryPoint` or `TA_InvokeCommandEntryPoint`). Then, we collect all descendants of the taint by recursively tainting all new definitions that depend on a tainted use. The propagation of the taint is dependent on the expression in which it is used. For instance, copying to a register, casting, or storing to memory introduces new taint flows. The taint can reach two kinds of sinks, a comparison or a function call. A comparison marks the containing CFG node as a type check node. A function call creates an entry in the worklist to later descend into the called function. We generate an alert for unexpected expressions. For instance, we do not expect the `paramTypes` to be used in arithmetic expressions or as a memory location that is written to or read from.

Tracking `params`. The intra-procedural analysis of `params` starts identical to the one performed for the `paramTypes` parameter. The taint is introduced by the corresponding parameter of one of the GP functions and points to an array of four `TEE_Param` unions. When collecting the descendants, we look for sinks that load data from the location of the first member of any of the entries in `params`, meaning a load of either `memref.buffer` or `value.a`. This preparation is typically expressed via (pointer) arithmetic where a (constant) offset is added to the base of the `params` array. Such a sink allows us to introduce a `param` taint for which we still need to determine if it is ever used as a `memref`. Similar to the `paramTypes` tracking, a usage of `params` as part of a function call creates a new worklist entry to later descend into the callee.

Tracking `param`. Each `param` taint represents a potential buffer. Hence, we collect all descendants for each `param` taint and proceed according to how these descendants are used. We mark a CFG node as `memref` usage sink if we encounter a taint being used as a reference to memory that is read from or written to. We add an item to the worklist containing the `param` taint when we encounter a call site consuming this taint as a parameter. This item will be processed later.

Alerting. Having the current function's CFG annotated with type check nodes and `memref` usage nodes, is a prerequisite to detect unchecked `memref` usages. For each `memref` usage, we determine if it is reachable without traversing a sanitizing type check node. If we can find such a path, GPCheck raises an alert, otherwise the usage is marked as correctly checked.

Inter-procedural Tracking. Similarly, the type check nodes help us to decide if we need to analyze functions contained in the worklist. If we can reach a call site without traversing a type check node, we have to analyze it. If a call site is only reachable by traversing a type check node, we discard the entry. The call-site-based analysis is context sensitive in the sense that the callee is analyzed multiple times depending on the call sites that propagate taints into it.

6 Implementation

We implemented GPCheck on top of Ghidra using the Ghidrathon plugin to enable Python3 support. GPCheck consists of roughly 2,100 lines of Python3 code. GPCheck leverages Ghidra's headless mode and can automatically analyze proprietary TAs of all major TEE implementations.

GPCheck performs a conservative static information flow analysis in the sense that alerts are raised for undetermined taint propagation situations. For instance if a taint is propagated into an external unknown function or an indirectly called function, GPCheck will warn the analyst that the taint can no longer be tracked. Our prototype expects a TA's code to behave well and reports cases of misbehavior. For instance, GPCheck will report if the `uint32_t paramTypes` parameter is used as a memory location. As an optimization, GPCheck maintains a catalog of known functions to shortcut the analysis. For instance, if a `param` taint is passed to a `memcpy()` function as `src` or `dst`, the call site is directly marked as a `memref` usage node instead of descending into the function. All the details of our taint propagation policies can be found in our open-source prototype.

7 GlobalConfusion Prevalence Study

In our evaluation, we aim to answer the following research questions:

RQ 1: GPCheck Effectiveness. Can GPCheck detect type-confusion bugs resulting from the design weakness of the GP Internal Core API specification in closed-source TAs?

RQ 2: Bug Class Prevalence. How prevalent are manifestations of the GP Internal Core API design weakness in contemporary TEE implementations?

RQ 3: Bug Class Severity. How severe is the type-confusion bug in real-world settings?

First, we focus on the effectiveness of GPCheck. We evaluate its GP function detection and type-confusion bug detection capabilities on a ground-truth dataset of proprietary real-world TAs. Next, we employ GPCheck to conduct a large-scale study to understand the prevalence of the type-confusion bug class within the TEE ecosystem. Then, we demonstrate the severity of this bug class by exploiting two discovered vulnerabilities under real-world conditions.

7.1 TA Dataset

We collected a dataset of 545 firmware images from 54 Android devices employing 5 different TEE implementations and spanning from 2016 to 2024. Our dataset is comprised of the top five Android Smartphone vendors covering a market share of over 60% [53]. From these firmware images we extracted 14,777 proprietary TAs (i.e., no open-source TAs), which grouped by their Universally Unique IDentifier (UUID) for each TEE implementation result in 374 TAs. While the TA grouping by UUID and TEE yields insights regarding the distribution and scope of our findings, it does not account for UUIDs being shared across TEEs and vendors assigning different UUIDs for TAs originating from the same code base. A manual data cleansing revealed that TA code is shared across BeanPod, Kinibi, and TEEGRIS on MediaTek chipsets. While the UUID assignment for Kinibi and BeanPod TAs is consistent, we found that it diverges for TEEGRIS. Our interpretation is that Samsung (the only vendor using TEEGris on MediaTek chips) assigned new UUIDs to these TAs. To account for these duplicates, we report the manually deduplicated aggregate number of TAs (#Unique TAs = 336) consistently in our evaluation. Our dataset is summarized in Table 2.

7.2 Ground-truth GP Function Detection

To assess GPCheck's ability to detect GP functions, we select three GP-compliant and three legacy TAs for each TEE implementation, since each of these implementations is using a different TA SDK. We select these TAs by randomly sampling from all TAs grouped by TEE and manually confirm the presence (or abscence) of the `TA_InvokeCommandEntryPoint` function. We conduct this experiment only for MiTEE, Kinibi

TEE	#TAs	#TA UUIDs
BeanPod	1,061	25
MiTEE	13	13
QSEE	7,798	189
Kinibi	1,316	67
TEEGRIS	4,589	80
Total	14,777	374 (336*)

Table 2: Our dataset of TAs for different TEE implementations. *Number of unique TAs.

and QSEE, because we found that the TAs running on the other TEEs (BeanPod and TEEGRIS) do not strip the `TA_InvokeCommandEntryPoint` symbol. Hence, we do not need the GP function detection step for these TAs. Our results are summarized in Table 3. GPCheck detected the GP API functions in all of the GP-compliant TAs in this evaluation. The detection of GP API functions is a prerequisite for the GP API type-confusion bug detection. Thus, our empirical evaluation partially answers RQ1.

TA	TEE	Stripped	GP-Compliant	Detected
vtrust.elf	QSEE	✓	✓	✓
eid.mbn	QSEE	✓	✓	✓
secstore2.elf	QSEE	✓	✓	✓
tz_kg.elf	QSEE	✓		
alipay.elf	QSEE	✓		
eseservi.elf	QSEE	✓		
655a4b46-cd77-11ea-aafbf382a6988e7b.ta*	MiTEE	✓	✓	✓
e97c270e-a5c4-4c58-bcd3384a2fa2539e.ta*	MiTEE	✓	✓	✓
14b0aad8-c011-4a3f-b66aca8d0e66f273.ta*	MiTEE	✓	✓	✓
05060000000000000000000000009578.tabin	Kinibi	✓	✓	✓
a2d1038963f25c97be80d40f2d498582.tabin	Kinibi	✓	✓	✓
09010000000000000000000000000000.tabin	Kinibi	✓	✓	✓
05120000000000000000000000000001.tlbin	Kinibi	✓		
030c0000000000000000000000000000.tlbin	Kinibi	✓		
07150000000000000000000000000000.tlbin	Kinibi	✓		

Table 3: We evaluate GPCheck's GP function detection capabilities on a ground-truth dataset of proprietary TAs. (*)We did not find any non-GP compliant MiTEE TAs in our dataset.

7.3 Ground-truth Type Checking

To assess GPCheck's ability to detect GP API type-confusion bugs, we collected a dataset of known vulnerable TAs. We obtained this information from publicly available advisories and blog posts of security researchers. Note that there are no open-source TAs in our dataset. Table 7 in our Appendix lists the sources for this ground truth.

As our results in Table 4 show, GPCheck is capable of detecting all known type-confusion bugs. Our empirical evaluation suggests that GPCheck is effective in detecting GP API type-confusion bugs in closed-source TAs (RQ1).

Vuln TA	TEE	Detected
task_storage	TrustedCore*	✓
d78d338b1ac349e09f65f4efe179739d.ta	BeanPod	✓
00000000-0000-0000-0000-000000000046	TEEGRIS	✓
00000000-0000-0000-0000-000048444350	TEEGRIS	✓
00000000-0000-0000-0000-0000534b504d	TEEGRIS	✓
00000000-0000-0000-0000-00575644524d	TEEGRIS	✓
00000000-0000-0000-0000-42494f535542	TEEGRIS	✓
00000000-0000-0000-0000-46494e474502	TEEGRIS	✓
00000000-0000-0000-0000-5345435f4652	TEEGRIS	✓
00000000-0000-0000-0000-53454d655345	TEEGRIS	✓

Table 4: We evaluate GPCheck's GP API type-confusion bug detection on a ground-truth dataset of vulnerable and proprietary TAs. (*) Huawei's TrustedCore is deprecated, but we found an old firmware image containing this publicly-known vulnerable TA [54].

7.4 Large-scale Study on Type Checking

In this section, we aim to measure the prevalence of the GP API type-confusion bug class on contemporary TEE implementations. Table 5 summarizes the results of our large-scale study. Overall GPCheck analyzed 14,777 TAs, which are comprised of 374 TAs with distinct UUIDs on five different TEEs (336 after deduplication). GPCheck detected 6,962 GP-compliant TAs mapping to 165 (43%) TAs, or 131 (35%) unique TAs, respectively. All GP-non-compliant TAs are found on Kinibi and QSEE. Kinibi and QSEE support their own proprietary APIs that were in use before the GP APIs were adopted. Thus, many of their legacy TAs continue to use these proprietary APIs.

We analyzed all 6,962 GP-compliant TAs using GPCheck. On average, the analysis of one TA took three minutes. We ran our experiments on a Xeon E5-2680 (56 cores, 256GB RAM) using 46 analysis jobs in parallel. Analyzing all 6,962 GP-compliant TAs takes about seven hours.

GPCheck found 850 vulnerable TAs in total. As a vulnerability deduplication step, we group these TAs by UUID and manually analyze the oldest and newest instances (two TAs per group). Overall this amounts to 86 manually analyzed TAs. We find that 8 TAs are flagged incorrectly as vulnerable by GPCheck (false-positive rate of 9.3%), due to imprecisions in Ghidra's decompilation and taints propagated into indirectly called functions that end up using the taint as `value` parameter. The remaining 78 TAs are true positives, resulting in 39 vulnerable and 33 unique vulnerable TAs. Table 6 shows the involved and affected parties of the vulnerable TAs. Note that about half of the vulnerable TAs in our dataset affect multiple TEE implementations or OEMs.

In total, we found 14 unique zero-day vulnerabilities (affecting the latest versions of TAs, see Table 8 in the Appendix) and 19 unique patched n-day vulnerabilities. Note that only 9 of the total 19 n-day type-confusion bugs are publicly known. The remaining n-day bugs were silently fixed by the vendor,

TEE	#TAs	#GP TAs	#Vuln	#TA UUIDs	#GP TA UUIDs	#Unique Vuln	#n-day	#0-day
BeanPod	1,061	1,061	277	25	25	11	4	7
MiTEE	13	13	1	13	13	1	0	1
QSEE	7,798	676	22	189	19	4	2	2
Kinibi	1,316	623	259	67	28	10	3	7
TEEGRIS	4,589	4,589	291	80	80	17	10	7
Total	14,777	6,962	850	374 (336)*	165 (131)*	39 (33)*	19 (19)*	24 (**14**)*

Table 5: The results of our large-scale study. The GP TAs are TAs that our analyzer has identified as utilizing the GlobalPlatform API. *The numbers in brackets account for TA code shared across TEEs and represent the unique count for these columns.

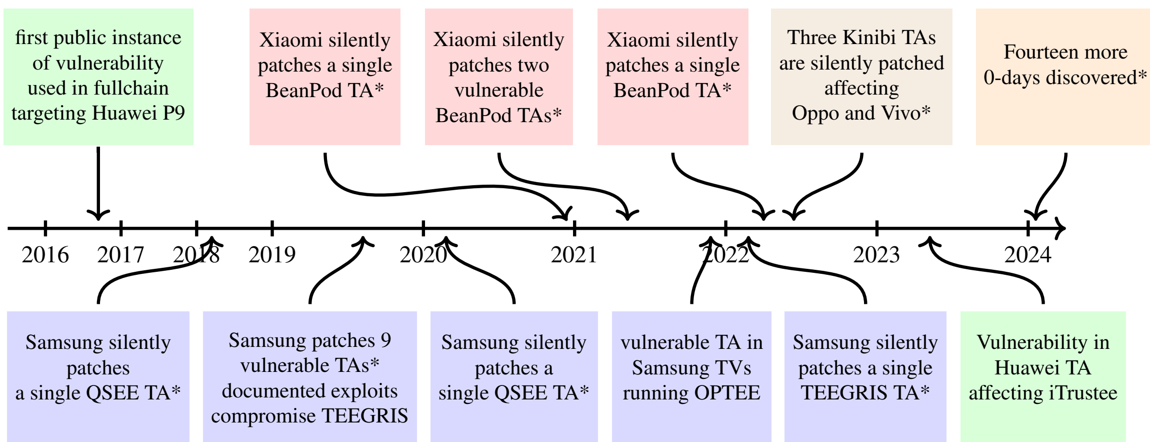

Figure 4: The timeline of known and discovered GP type-confusion bug instances in production TAs. These bugs affect TAs deployed on products by Huawei, Xiaomi, Oppo and Vivo, Samsung, and undisclosed vendors. (*)Type-confusion bugs discovered by GPCheck.

meaning that the affected TAs were vulnerable prior to these silent fixes.

A further aspect of this prevalence study is the perspective of TA developers. Our study reveals that TA developers missed the type check in GP-compliant TAs in 33 out of 131 cases (23%), as indicated in Table 5.

The prevalence of type-confusion vulnerabilities resulting from the GP Internal Core API design weakness systematically occur throughout the history of TAs. We visualize our findings of known, silently fixed, and 0-day vulnerabilities in Figure 4. Over the past seven years, TA developers stepped into the design-weakness trap of this API over and over again. As Table 6 suggests, we see the resulting vulnerabilities in TAs across the major OEMs on the mobile device market, across TEE implementations, and ODMs (Orignal Design Manufacturer). Our results show a wide prevalence of the vulnerabilities in the mobile device ecosystem, which answers RQ2.

7.5 Real-world Exploitation

To show the severity of the type-confusion bug, we exploit two vulnerable TAs that we detected with our static analyzer. We

ODM	OEM	TEEs	#TAs
MediaTek	Samsung,Xiaomi,Oppo,Vivo	TEEGRIS,Kinibi,BeanPod	6
Exynos	Samsung	TEEGRIS	14
MediaTek,Qualcomm	Xiaomi	BeanPod,QSEE,MiTEE	7
MediaTek	Vivo,Oppo	Kinibi	3
MediaTek	Vivo	Kinibi	2
Qualcomm	Samsung	QSEE	1

Table 6: #Unique vulnerable TAs and affected parties.

show how the type-confusion bug can be used to easily leak the memory of a TA and how, under the right circumstances, this bug may be weaponized to get control of the program counter. We exploit these TAs on our rooted Xiaomi Redmi device with the newest firmware version.

TA1449 (`14498ace2a8f11e880c8509a4c146f4c.ta`) was silently patched by Xiaomi in April 2021. Although the TA is outdated we can load it on our fully updated firmware due to missing rollback protection in BeanPod [41]. Listing 4 shows the vulnerable code. In the code that handles the command id `1`, the TA expects a memory reference for the first parameter. It then reads a string from this location and prints it to the kernel log which is accessible from the Normal World context. By supplying arbitrary integers, which are

```
TA_InvokeCommandEntryPoint(void* sessionContext,
    uint32_t commandID, uint32_t paramTypes,
    TEE_Param params[4]) {
  ...
  if(commandID == 1) {
    ...
    char* aaid = (char*)params[0].memref.buffer;
    ...
    ut_pf_log_msg(3,"add Authenticator \
        aaid = %s\n",aaid);
  ...
  }
}
```

Listing 4: Without checking the `paramTypes` the TA dereferences the first entry of the parameters and prints it to the kernel log.

treated as pointers by the TA, an attacker can read the entire content of the TA's memory. This arbitrary read is a problem as the TA relies on the confidentiality of secret keys stored in its memory to authenticate certain commands.

For our second case study, we show how this bug can lead to code execution within the context of the targeted TA. We exploit the `08110000000000000000000000000000.ta` (TA0811) and get control over the TA's program counter. The type-confusion vulnerability in this TA was a 0-day detected by our static analyzer and has been assigned CVE-2023-32835 by MediaTek. It affects Xiaomi, Oppo, and Vivo smartphones with a MediaTek SoC running a firmware version from before November 2023.

Listing 5 shows the relevant vulnerable code executed when the TA is invoked with the command id `1`. The TA assumes, without checking, the first and second parameters to be pointers to shared memory. It subsequently passes the first of these supposed pointers to the `TEE_CheckMemoryAccessRights` function. This function checks if the memory region pointed to by `in` points to readable and writable memory. By setting `in` to an arbitrary value and observing if the TA returns early or executes the `query_drmkey_impl` function, the attacker can leak the location of relevant memory regions, such as the stack. After leaking the stack's location, the attacker can use the functionality in `query_drmkey_impl` to overwrite the return address. This function takes as input the first and second memory reference in `params`. It then writes data from the first parameter to the second parameter. The attacker can set `out` to point to the stack, specifically to the location of the stored return address. The TA then writes the contents in `in`, which are fully under the attacker's control, to the stack. In our exploit, we overwrite the return address with `0xdeadbeef`. After crashing, the TEE conviniently prints a core dump to the kernel log accessible from the Normal World context, indicating that the TA aborted when trying to execute code at `0xdeadbeef`. Note that this control flow hijacking primitive

```
TA_InvokeCommandEntryPoint(void* sessionContext,
    uint32_t commandID, uint32_t paramTypes,
    TEE_Param params[4]) {
  ...
  if(commandID == 1) {
    int* in = (int*)params[0].memref.buffer;
    int inSize = params[1].memref.size;
    int* out = (int*)params[2].memref.buffer;
    r = TEE_CheckMemoryAccessRights(5, in,
                                        inSize);
    if(r==0) {
      query_drmkey_impl(in, out);
    }
  }
  ...
}

query_drmkey_impl(int* in, int* out) {
  int ct = in[17];
  int* src = &in[18];
  int tmpbuf[22];
  int c;
  out[0] = ct;
  int i = 0;
  while(i != ct){
    memcpy(tmpbuf, src, 0x58);
    int offset = tmpbuf[3] + 0x60;
    c = tmpbuf[0];
    out[4*i] = c;
    src = src + offset;
    i++;
  }
  ...
 }
```

Listing 5: The vulnerable parts of the code in TA0811. The TA does not check the `paramTypes` and treats entries in the `params` array as pointers. This allows an attacker to enumerate read/writable memory regions using `TEE_CheckMemoryAccessRights`. Furthermore, the attacker can write to arbitrary memory addresses, abusing the TAs `query_drmkey_impl` function.

can easily be converted into arbitrary code execution within the TA context.

These case studies demonstrate that the ability of an attacker to supply arbitrary pointers to a TA provides an extremely powerful exploitation primitive, and answer our RQ3.

8 Mitigation

GPCheck serves two purposes. First, it enabled us to carry out a large-scale study to understand the prevalence of type-confusion bugs resulting from the GP Internal Core API design weakness. Second, we will open-source our prototype and encourage manufacturers to use it as a vetting mechanism for post-production TAs before they are shipped to consumer devices. Given the results of our evaluation in Section 7, showing 14 0-day vulnerabilities, we propose to not only treat the symptoms of these vulnerabilities but also provide a remedy for their root cause.

To mitigate the GP Internal Core API design weakness, we suggest extending the specification and substituting the optional type checking mechanism for a fail-safe alternative. This extension aims at enforcing the currently optional type check before any untrusted parameter reaches the TA. The specification contains two API functions that receive untrusted parameters and their types, namely `TA_OpenSessionEntryPoint` and `TA_InvokeCommandEntryPoint`. Looking at hundreds of proprietary TA samples and those provided by OPTEE yields the insight that the `cmdId`[1] chooses the corresponding handler and, the code of this handler determines how the four `params` are used, namely either as `value` or as `memref`. Consequently, it is the tuple of `cmdId` and `paramTypes` that needs to be fixed and enforced before any untrusted input is supplied to TAs.

From this insight, we propose to deny all `cmdIds` and the invocation of `TA_OpenSessionEntryPoint` by default, and leverage the `TA_CreateEntryPoint` lifecycle function to register command handler-specific `paramTypes`. As discussed in Section 2, `TA_CreateEntryPoint` acts as a constructor and is executed before any untrusted parameters can be sent to the TA. To register the `(cmdId, paramTypes)` tuples, we introduce two new functions to be used in the `TA_CreateEntryPoint` context. Listing 6 illustrates the signatures of `TEE_RegisterCommand`, to register `(cmdId, paramTypes)` tuples for the `TA_InvokeCommandEntryPoint` function, and `TEE_RegisterOpenSession`, to register the `paramTypes` for the single command handler in the `TA_OpenSessionEntryPoint` function. By denying access to any unregistered interface that consumes untrusted data, and enforcing a deliberate registration of `paramTypes` associated with a specific command handler, we change the current fail-open design to a fail-closed one. As a consequence, we prevent any mistakenly forgotten type checks in future implementations, and mitigate the design weakness of the current GP Internal Core API specification.

```
TEE_Result TEE_RegisterCommand(
    uint32_t cmdId, uint32_t paramTypes);

TEE_Result TEE_RegisterOpenSession(uint32_t paramTypes);
```

Listing 6: We suggest to extend the GP Internal Core API specification with functions to enforce a fail-closed rather than a fail-open design.

[1] `TA_OpenSessionEntryPoint` does not receive this parameter and does not contain any `cmdId`-based switching logic. Hence, we can assume a single fixed set of `paramTypes` for this function.

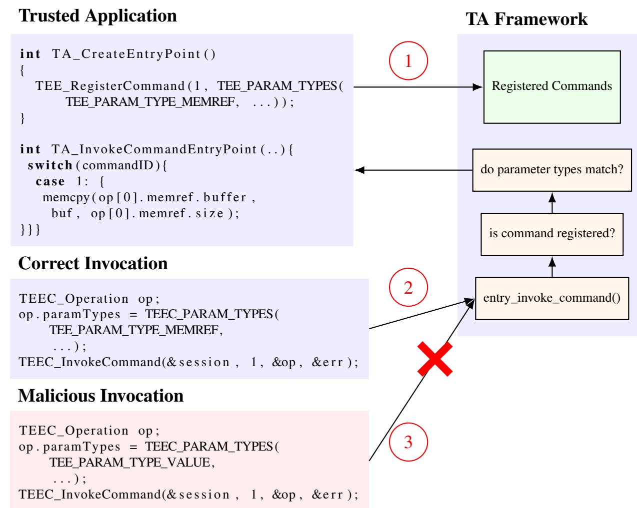

Figure 5: Inner workings of our mitigation.

Figure 5 illustrates the inner workings of our mitigation. First, ① the TA registers a single command with `cmdId` 1 to have `paramTypes` `TYPE_MEMREF`. After the TA has been loaded, the normal world may invoke the registered commands ②. Such invocations are handled in the TA framework (e.g., `entry_invoke_command` function), which eventually calls the TA's `TA_InvokeCommandEntryPoint` function. However, before the TA framework hands over execution to the TA, it verifies that the `cmdId` was previously registered by the TA and that the `paramTypes` parameter matches the types registered by the TA. An example of the mitigation preventing an attack is shown in Figure 5. In ③, the attacker has manipulated the parameter type to be `TYPE_VALUE` and the TA without checking the `paramTypes` would have treated these values as pointers. However, due to the TA registering the `cmdId` 1 with `TYPE_MEMREF`, the TA framework will deny this API invocation, since the `paramTypes` of the invocation does not match with the expected types.

Compatibility. Our mitigation is backward-compatible with existing TAs in the sense that the exposed TA interface stays untouched. However, changing from a fail-open to a fail-closed design requires TA developers to register `cmdIds` for all existing TAs in a one-time effort. Given the reccuring pattern of bugs within the last seven years, as illustrated in Figure 4, and the 14 0-day vulnerabilities discovered in this research, we argue that this one-time effort is a worthwhile investment to eliminate present and future GP API-based type-confusion bugs once and for all.

Implementation and Evaluation. To demonstrate the effectiveness and practicality of our mitigation, we implement our extension for OPTEE [59], the de-facto reference implementation for TrustZone-based TEEs. In order to demonstrate the effectiveness of our mitigation, we modify the built-in TA `pkcs11` to remove `paramTypes` checks. Since this TA is using `memref` typed parameters, an attacker can pass arbitrary pointers to the TA and gain powerful read-and-write exploitation primitives when `pkcs11` runs on vanilla OPTEE. In contrast, after enabling our mitigation, passing a wrong combination of `paramTypes` and `cmdId` to the TA results in a `TEE_ERROR_BAD_PARAMETERS` error, effectively preventing the type confusion. We implement the mitigation in 427 lines of code and will open-source our evaluation setup.

9 Limitations

Encrypted TAs. Recent firmware images from vendors like Huawei and Sony are encrypted and cannot directly be processed by our analysis pipeline. Breaking this encryption and, thus, the code confidentiality of encrypted TAs is an orthogonal problem. However, these vendors themselves might be interested to use GPCheck in their integration pipeline, or eager security researchers might break the code confidentiality for individual devices and then use GPCheck.

Decompilation. Compilation is a lossy process. Thus, the inverse process, decompilation, relies on heuristics and inference methods to re-create structures from the original source code. GPCheck relies on commodity reverse-engineering tools and their intermediate representations used in the decompilation process. Consequently, our system inherits the limitations of decompilation, and the specific limitations of Ghidra, since we build GPCheck on top of this reverse-engineering tool. Fortunately, we did not encounter any obfuscation, the dominant programming language to implement TAs is C, and the type checking problem investigated in this study is fairly narrow. These factors contribute to meaningful decompilation results and useful static analysis results.

10 Discussion

In this section, we discuss the findings of our work and contextualize their scope and impact. First, we argue for a fail-safe design in widely adopted TEE-related specifications. Second, we emphasize the impact and threat potential of our findings. Third, we give an outlook regarding the future adoption of the GP Internal Core API specification.

Enforcing Fail-Safe Design. In its current form, the GP Internal Core API specification proposes a weak design that does not enforce type checks of untrusted parameters. The specification suggests an *optional* preprocessor macro-based type check. This lack of "fail-safe" design resulted in 19 n-day vulnerabilities (9 publicly known and 10 silently fixed) and 14 0-day vulnerabilities (found by our system GPCheck) affecting the latest versions of TAs. These numbers underline that especially in a sensitive context like the TEE, fail-safe design principles should be followed, and we must enforce security-critical type checks. In this context, GPCheck serves two purposes. First, it allowed us to assess the prevalence of manifestations of the GP API design weakness, identifying this weakness as a serious threat to the TEE ecosystem. Second, it can serve manufacturers as a post-production vetting tool to catch vulnerable TAs before they are shipped to devices of billions of users. However, instead of treating the symptoms, our backwards-compatible mitigation proposed in Section 8, aims to eliminate the root cause of these vulnerabilities by enforcing type checks and substituting the current weak design for a fail-safe one.

Keeping Promises. In total, we found 39 vulnerable TAs across 5 reputable OEMs affecting 54 recent devices that employ the 5 dominant TEE implementations on the market. Given that OEMs are struggling to deploy effective anti-rollback protection [10] (i.e., preventing to load properly signed but outdated and vulnerable TAs into their TEE), the threat posed by a single vulnerable TA in the history of a device is amplified. Combined with the critical severity [49] of the vulnerabilities resulting from the GP Internal Core API design weakness, the TEE ecosystem must change to keep up their promised confidentiality and integrity guarantees.

Outlook. Our dataset of 336 unique TAs deployed on a representative set of mobile devices in the ecosystem highlights the trend towards GP-compliance. Hence, we believe that the central piece of our discussion, the weak design of the GP Internal Core API, will be adopted by an increasing amount of stakeholders in the future. Further, our study focussed on mobile devices but the scope of this discussion reaches far beyond this device category. For instance, drones [15], TVs [45], tablets [50], and gaming consoles [57] require strong confidentiality and integrity guarantees for specific use cases and are using TrustZone-based TEEs for these purposes. Beyond these ARM-centric device categories, GP's specifications aspire to be architecture-agnostic and we might see them being adopted on other TEE-enabling technologies like Intel SGX, Intel TXT, AMD SEV, and ARM CCA.

11 Related Work

Related work exists across several domains. First, we cover TrustZone-based TEE flaws in general. Second, we highlight approaches to find or prevent memory corruptions in TEEs. Third, we summarize measurement studies in the mobile ecosystem. Finally, we relate to existing work on binary static analysis.

TEE Flaws. Numerous researchers have delved into the exploration and exploitation of vulnerabilities within TrustZone-based Trusted Execution Environments (TEEs). These studies have targeted various TEE implementations, such as Beanpod [41], QSEE [34, 40, 48], TEEGRIS [43, 51, 52], Kinibi [4, 7, 33], TrustedCore [11], and the ARM Trusted Firmware [37]. Furthermore, researchers have scrutinized software design deficiencies [39, 55] and side-channel vulnerabilities [8, 35, 58, 61]. A comprehensive overview of much of this research has been provided by Cerdeira et al. [12], who summarized and systematized these findings.

Memory Corruption Defenses. The increasing incidence of reported memory corruptions has spurred research efforts towards automated vulnerability discovery within TEEs. TEEzz [9] employs an on-device black-box fuzzing strategy, while PartEmu [26] adopts an emulation-based approach to facilitate coverage-guided fuzzing of TAs. In contrast, Wan et al. endeavor to promote Rust as a memory-safe alternative for TA development [60], rather than focusing on automated memory corruption detection. Our work focuses on a specific vulnerability commonly found in GP-compliant TAs. We propose an effective binary static analysis tool, GPCheck, to find this vulnerability post-production and suggest eliminating the root cause of this issue by updating the GP Internal Core API using our mitigation that enforces a fail-safe design.

Measurement Studies. The widespread adoption of Android has attracted significant attention from security researchers, leading to comprehensive investigations into various facets of its security architecture. Imran et al. [27] measure the usage of TEE-enforced authorization APIs within apps. Farhang et al. [20] scrutinize Android security bulletins issued by different vendors. Numerous studies delve into the application of updates to apps or libraries [5, 14, 42]. Additionally, several investigations have concentrated on identifying and understanding diverse security vulnerabilities present in the Android ecosystem [19, 36]. Finally, Egele et al. [17] conduct an empirical analysis of how developers misuse cryptographic APIs within Android applications. In this paper, we measured the prevalence of type-confusion bugs resulting from a weak API design as proposed by the GP Internal Core API specification. In total, we found 33 unique TAs across all major TEEs that are affected by this critical vulnerability.

Static Analysis. Due to the hardware-enforced isolation of TEEs, vendors effectively lock down their platforms and prevent dynamic analysis approaches like advanced fuzzing provided by LibAFL [21] to discover vulnerabilities in TAs. Further, TAs are distributed as proprietary binary blobs, making them inaccessible for source code-based static analysis approaches as proposed by Yamaguchi et al. [62–64]. Binary static analysis approaches like Bootstomp [46], Bootkeeper [13], and Karonte [47] suggest domain-specific analyses for bootloaders and embedded firmware. These approaches often use additional techniques like dynamic symbolic execution to reduce reported false positives. Our system, GPCheck, concentrates on a well-defined problem in the domain of TAs. We leverage powerful decompilation features of commodity reverse-engineering tools and use the architecture-agnostic and optimized intermediate representation of these tools for our analyses.

12 Conclusions

A design weakness in the GlobalPlatform Internal Core API specification, a specification that serves as the de-facto standard for Trusted Applications, impacts the security of billions of mobile devices. Manifestations of this design weakness lead to critical type-confusion vulnerabilities that threaten the integrity and confidentiality guarantees promised by modern TEEs. After discovering this weakness, we investigate the prevalence of such vulnerabilities and design and implement GPCheck, a static information flow tracking system that is based on commodity reverse-engineering tools. GPCheck allowed us to carry out a large-scale analysis aimed at discovering instances of the type-confusion bug in real-world closed-source TAs that we obtained from firmware images of various popular mobile devices.

In total, we analyzed 14,777 TAs and found 33 instances of the type-confusion issue. 9 out of these bugs are publicly known, 10 were silently fixed by vendors, and 14 bugs were unknown 0-days that we responsibly disclosed to the affected manufacturers. These disclosures resulted in four CVEs and the remaining 10 critical vulnerabilities are still in the responsible disclosure process. Finally, we proposed a mitigation to eliminate the root cause of these type-confusion vulnerabilities. Our backward-compatible mitigation enforces the currently optional type check of untrusted TA parameters and only requires 427 additional lines of code when added to the OPTEE reference implementation for TrustZone-based TEEs. We suggested this mitigation to GlobalPlatform as an extension to the GP Internal Core API to hopefully substitute the design weakness for a fail-safe alternative. As a stop-gap solution, our open-source GPCheck prototype assists manufacturers in vetting their TAs before they are deployed on customer devices and bridges the time gap until our mitigation is adopted to eliminate the issue by design.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their feedback on the paper. This work was supported, in part, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2_186974, and DARPA HR001119S0089-AMP-FP-034.

References

[1] Vector 35. Binary ninja intermediate language: Overview. https://docs.binary.ninja/dev/bnil-overview.html, 2024. Accessed: January 2024.

[2] National Security Agency. P-code operation reference. https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/

html/pcodedescription.html, 2024. Accessed: January 2024.

[3] Maxime Peterlin Alexandre Adamski. Huawei trustzone tee_service_face_rec vulnerabilities. https://blog.impalabs.com/2309_advisory_huawei_trustzone_tee-service-face-rec.html, 2023. Accessed: January 2024.

[4] Maxime Peterlin Alexandre Adamski, Joffrey Guilbon. A deep dive into samsung's trustzone (part 1 - part 3). https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-1.html, 2019. Accessed: April 2023.

[5] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. Too quiet in the library: An empirical study of security updates in android apps' native code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1347–1359. IEEE, 2021.

[6] ARM. Arm security technology: Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008. Accessed: January 2024.

[7] David Berard. Kinibi tee: Trusted application exploitation. https://www.synacktiv.com/en/publications/kinibi-tee-trusted-application-exploitation.html, 2018. Accessed: April 2023.

[8] Sébanjila Kevin Bukasa, Ronan Lashermes, Hélène Le Bouder, Jean-Louis Lanet, and Axel Legay. How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip. In *Workshop in Information Security Theory and Practice*, 2017.

[9] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. Teezz: Fuzzing trusted applications on cots android devices. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 220–235. IEEE Computer Society, 2022.

[10] Marcel Busch, Philipp Mao, and Mathias Payer. Spill the tea: An empirical study of trusted application rollback prevention on android smartphones. In *33st USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[11] 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, August 2020.

[12] 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, San Francisco, CA, USA, May 18-21, 2020*, pages 1416–1432. IEEE, 2020.

[13] Ronny Chevalier, Stefano Cristalli, Christophe Hauser, Yan Shoshitaishvili, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna, Danilo Bruschi, and Andrea Lanzi. Bootkeeper: Validating software integrity properties on boot firmware images. In Gail-Joon Ahn, Bhavani Thuraisingham, Murat Kantarcioglu, and Ram Krishnan, editors, *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, pages 315–325. ACM, 2019.

[14] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[15] DJI. System security – a dji technology white paper. https://djiarsmadrid.com/pdfdoc/DJI%20Security%20White%20Paper.pdf, 2020. Accessed: January 2024.

[16] Gregory J. Duck and Roland H. C. Yap. Effectivesan: type and memory error detection using dynamically typed C/C++. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 181–195. ACM, 2018.

[17] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84, 2013.

[18] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick D. McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 393–407. USENIX Association, 2010.

[19] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities, 2019.

[20] Sadegh Farhang, Mehmet Bahadir Kirdan, Aron Laszka, and Jens Grossklags. An empirical study of android security bulletins in different vendors. In *Proceedings of The Web Conference 2020*, WWW '20, page 3063–3069, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, CCS '22. ACM, November 2022.

[22] GlobalPlatform. Tee internal core api specification. https://globalplatform.org/specs-library/tee-internal-core-api-specification/, 2019. Accessed: January 2024.

[23] Google. Hardware-backed keystore. https://source.android.com/docs/security/features/keystore, 2024. Accessed: January 2024.

[24] Ilfak Guilfanov. Decompiler internals: Microcode. https://i.blackhat.com/us-18/Thu-August-9/us-18-Guilfanov-Decompiler-Internals-Microcode-wp.pdf, 2018. Accessed: January 2024.

[25] István Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 517–528. ACM, 2016.

[26] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. Partemu: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[27] Abdullah Imran, Habiba Farrukh, Muhammad Ibrahim, Z. Berkay Celik, and Antonio Bianchi. SARA: secure android remote authorization. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 1561–1578. USENIX Association, 2022.

[28] Yuseok Jeon, Priyam Biswas, Scott A. Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for C++. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2373–2387. ACM, 2017.

[29] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 258–263. IEEE Computer Society, 2006.

[30] Stephen Kell. Dynamically diagnosing type errors in unsafe code. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 800–819. ACM, 2016.

[31] Dongju Kim and Seungjoo Kim. Bintyper: Type confusion detection for c++ binaries. https://i.blackhat.com/eu-20/Thursday/eu-20-Kim-BinTyper-Type-Confusion-Detection-For-C-Binaries.pdf, 2020. Accessed: May 2024.

[32] Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. Uncontained: Uncovering container confusion in the linux kernel. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 5055–5072. USENIX Association, 2023.

[33] kutyacica. Unbox your phone (part 1 - part 3). https://labs.taszk.io/articles/post/unbox_your_phone_1/, 2018. Accessed: April 2023.

[34] laginimaineb. Exploring qualcomm's secure execution environment. http://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html, 2016. Accessed: April 2023.

[35] Paul Leignac, Olivier Potin, Jean-Baptiste Rigaud, Jean-Max Dutertre, and Simon Pontié. Comparison of side-channel leakage on rich and trusted execution environments. In *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, CS2 '19, page 19–22, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 2–13, 2017.

[37] Christian Lindenmeier, Mathias Payer, and Marcel Busch. El3xir: Fuzzing cots secure monitors. In *33st USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[38] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[39] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017.

[40] Slava Makkaveev. The road to qualcomm trustzone apps fuzzing. https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/, 2019. Accessed: April 2023.

[41] Slava Makkaveev. Researching xiaomi's tee to get to chinese money. https://research.checkpoint.com/2022/researching-xiaomis-tee/, 2022. Accessed: April 2023.

[42] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21:1346–1370, 2016.

[43] Frederico Menarini. Breaking tee security part 1-part 3. https://www.riscure.com/tee-security-samsung-teegris-part-1/, 2019. Accessed: April 2023.

[44] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. Mapping to bits: Efficiently detecting type confusion errors. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 518–528. ACM, 2018.

[45] raelize. Vulnerable tzdemuxerservice ta on samsung tvs (j-series). https://raelize.com/blog/samsung-tv-tzdemuxerservice-ta-vulnerabilities/, 2021. Accessed: January 2024.

[46] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. BootStomp: On the security of bootloaders in mobile devices. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 781–798, Vancouver, BC, August 2017. USENIX Association.

[47] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1544–1561. IEEE, 2020.

[48] Dan Rosenberg. Reflections on trusting trustzone. Black Hat 2014, 2014. Accessed: April 2023.

[49] Samsung. Samsung vulnerability bulletin. https://security.samsungmobile.com/securityUpdate.smsb, 2019. Advisories from August 2019 (SVE-2019-14892, SVE-2019-14891, SVE-2019-14885, SVE-2019-14885, SVE-2019-14864, SVE-2019-14851, SVE-2019-14850, SVE-2019-14847). Accessed: January 2024.

[50] Samsung. Security configuration guide samsung galaxy mobile and tablet devices. https://kp4-cdn.samsungknox.com/resource/Samsung%20Mobile%20Device%20-%20Security%20Configuration%20Guide%208%20September%202023_CXAN.pdf, 2023. Accessed: January 2024.

[51] Eloi Sanfelix. Tee exploitation - exploiting trusted apps on samsung's tee. https://downloads.immunityinc.com/infiltrate2019-slidepacks/eloi-sanfelix-exploiting-trusted-apps-in-samsung-tee/TEE.pdf, 2019. Accessed: April 2023.

[52] Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's TrustZone keymaster design. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 251–268, Boston, MA, August 2022. USENIX Association.

[53] Statista. Global smartphone market share from 4th quarter 2009 to 2nd quarter 2023. https://www.statista.com/statistics/271496/global-market-share-held-by-smartphone-vendors-since-4th-quarter-2009/, 2023. Accessed: October 2023.

[54] Nick Stephens. Behind the pwn of a trustzone. https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose, 2016. Accessed: April 2023.

[55] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal privilege escalation in trusted applications. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[56] Mingshen Sun, Tao Wei, and John C. S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 331–342. ACM, 2016.

[57] switchbrew.org. Switch system flaws. https://switchbrew.org/wiki/Switch_System_Flaws#TrustZone, 2023. Accessed: January 2024.

[58] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Clkscrew: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, volume 2, pages 1057–1074, 2017.

[59] TrustedFirmware.org. Op-tee documentation - trusted applications. https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html, 2023. Accessed: April 2023.

[60] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe ARM trustzone applications. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 442–453. ACM, 2020.

[61] Jie Wang, Kun Sun, Lingguang Lei, Shengye Wan, Yuewu Wang, and Jiwu Jing. Cache-in-the-middle (citm) attacks: Manipulating sensitive data in isolated execution environments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1001–1015, 2020.

[62] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 590–604. IEEE Computer Society, 2014.

[63] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 797–812. IEEE Computer Society, 2015.

[64] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 499–510. ACM, 2013.

[65] Yuxing Zhang, Xiaogang Zhu, Daojing He, Minhui Xue, Shouling Ji, Mohammad Sayad Haghighi, Sheng Wen, and Zhiniang Peng. Detecting union type confusion in component object model. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4265–4281. USENIX Association, 2023.

13 Appendix

Vuln TA	Name	CVE	Source
task_storage	Secure Storage	CVE-2016-8764	[54]
d78d338b1ac349e09f65f4efe179739d.ta	Soter	None	[41]
00000000-0000-0000-0000-000000000046	MLDAP	SVE-2019-14867	[43, 49]
00000000-0000-0000-0000-000048444350	HDCP	SVE-2019-14850	[43, 49]
00000000-0000-0000-0000-0000534b504d	SKPM	SVE-2019-14892	[43, 49]
00000000-0000-0000-0000-00575644524d	WVDRM	SVE-2019-14885	[43, 49]
00000000-0000-0000-0000-42494f535542	EXT_FR	SVE-2019-14847	[43, 49]
00000000-0000-0000-0000-46494e474502	FINGERPRINT	SVE-2019-14864	[43, 49]
00000000-0000-0000-0000-5345435f4652	SEC_FR	SVE-2019-14851	[43, 49]
00000000-0000-0000-0000-53454d655345	SEM	SVE-2019-14891	[43, 49]

Table 7: Our ground-truth dataset of vulnerable TAs.

0-Day TA	Disclosure Status
gpeid	Confirmed
chnactiv	Confirmed
08110000000000000000000000000000	CVE-2023-32835
08030000000000000000000000000000	CVE-2023-32834
06140000000000000000000000000000	CVE-2023-32848
09010000000000000000000000000000	Confirmed
08020000000000000000000000007169	Confirmed
09030000000000000000000000008270	Confirmed
98fb95bcb4bf42d26473eae48690d7ea	Confirmed
abcd270ea5c44c58bcd3384a2fa2539e	Confirmed
07770000000000000000000000000000	Confirmed
e97c270e-a5c4-4c58-bcd3384a2fa2539e	Confirmed
00000000-4d54-4b5f-4246-566964456e63	CVE-2024-20078
00000000-4d54-4b5f-4246-564448455643	Confirmed

Table 8: 0-Day vulnerabilities discovered in our study.