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 millions 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.

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 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.

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 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.

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 TrustZone-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 initializing 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.

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 vulnerabilities.

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

Listing 2: An echo-TA to demonstrate how the two affected GP API functions are used.
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 {
        value; 
    } value;
    union {
        struct {
            value;
        } value;
        struct {
            in_buf, out_buf;
            size_t in_buf_sz, out_buf_sz;
        } memref;
        struct {
            size_t in_buf_sz, out_buf_sz;
            in_buf = (char*)params[0].memref.buffer;
            out_buf = (char*)params[1].memref.buffer;
        } memref;
    } memref;
} TEE_Param;
```

Listing 3: The parameter type is either a 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.

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

```c
1 #define TEE_PARAM_TYPE_NONE 0
2 #define TEE_PARAM_TYPE_VALUE_INPUT 1
3 #define TEE_PARAM_TYPE_VALUE_OUTPUT 2
4 #define TEE_PARAM_TYPE_MEMREF_INPUT 5
5 #define TEE_PARAM_TYPE_MEMREF_OUTPUT 6
6 #define TEE_PARAM_TYPE_MEMREF_INPUT 5
```
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
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 temporary value elimination, common subexpression elimination, and other compiler optimizations. This optimized IR can then be analyzed to detect type-confusion vulnerabilities.

## 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_Param is 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 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_Props may be used as a memref. In particular, any access (dereference) of the memref member of TEE_Param 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 ac-

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

<table>
<thead>
<tr>
<th>Solution</th>
<th>Analysis</th>
<th>Target</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeScan [25]</td>
<td>Dynamic</td>
<td>Source (C++)</td>
<td>Runtime Sanitization</td>
</tr>
<tr>
<td>HexType [28]</td>
<td>Dynamic</td>
<td>Source (C++)</td>
<td>Runtime Sanitization</td>
</tr>
<tr>
<td>BrType [44]</td>
<td>Dynamic</td>
<td>Source (C++)</td>
<td>Runtime Sanitization</td>
</tr>
<tr>
<td>ещiveness [16]</td>
<td>Dynamic</td>
<td>Source (C++)</td>
<td>Runtime Sanitization</td>
</tr>
<tr>
<td>Uncontexted [32]</td>
<td>Hybrid</td>
<td>Source (C)</td>
<td>Def-Use Chains / Runtime Sanitization</td>
</tr>
<tr>
<td>TaintDroid [18]</td>
<td>Dynamic</td>
<td>Bytecode (Dalvik)</td>
<td>Runtime Taint Tracking</td>
</tr>
<tr>
<td>TaintArt [58]</td>
<td>Dynamic</td>
<td>Bytecode (Dalvik)</td>
<td>Runtime Taint Tracking</td>
</tr>
<tr>
<td>libcrunch [30]</td>
<td>Dynamic</td>
<td>Binary (C++)</td>
<td>Runtime Sanitization</td>
</tr>
<tr>
<td>BinType [31]</td>
<td>Dynamic</td>
<td>Binary (C++)</td>
<td>Class Recovery / Runtime Sanitization</td>
</tr>
<tr>
<td>Bootstomp [46]</td>
<td>Static</td>
<td>Binary (C)</td>
<td>Static Taint Propagation</td>
</tr>
<tr>
<td>Karoo [47]</td>
<td>Static</td>
<td>Binary (C)</td>
<td>Static Taint Propagation</td>
</tr>
<tr>
<td>COMfusion [65]</td>
<td>Static</td>
<td>Binary (C)</td>
<td>Static Taint Propagation</td>
</tr>
</tbody>
</table>

### 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.
cesses 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 over-tainting. 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-positive 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
We implemented GPCheck on top of Ghidra using the Ghidra 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 to 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 absence) of the TA_InvokeCommandEntryPoint function. We conduct this experiment only for MiTEE, Kinibi
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.

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.

<table>
<thead>
<tr>
<th>TEE</th>
<th>#TAs</th>
<th>#TA UUIDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanPod</td>
<td>1,061</td>
<td>25</td>
</tr>
<tr>
<td>MiTEE</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>QSEE</td>
<td>7,798</td>
<td>189</td>
</tr>
<tr>
<td>Kinibi</td>
<td>1,316</td>
<td>67</td>
</tr>
<tr>
<td>TEEGRIS</td>
<td>4,589</td>
<td>80</td>
</tr>
<tr>
<td>Total</td>
<td>14,777</td>
<td>374 (336*)</td>
</tr>
</tbody>
</table>

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].

<table>
<thead>
<tr>
<th>Vuln TA</th>
<th>TEE</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_storage</td>
<td>TrustedCore*</td>
<td>✓</td>
</tr>
<tr>
<td>d78a338b1ac349e0916f644e179739d.ta</td>
<td>BeanPod</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-000000000046</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-0000048444350</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-0000534b504d</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-000057564524</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-00004294953542</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-46494e474502</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-534543465652</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-534546553454</td>
<td>TEEGRIS</td>
<td>✓</td>
</tr>
</tbody>
</table>

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).

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 tainted value.

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

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.

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 (Original 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 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
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 conveniently 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 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, show-
such invocations are handled in the TA frame-
work. This implies that the TA registers a single command with the TA framework,
and that the 

\text{cmdId}\text{ 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 \text{TA\_OpenSessionEntryPoint} by default, and leverage the \text{TA\_CreateEntryPoint} lifecycle function to register command handler-specific paramTypes. As discussed in Section 2, \text{TA\_CreateEntryPoint} acts as a constructor and is executed before any untrusted parameters can be sent to the TA. To register the \text{(cmdId, paramTypes)} tuples, we introduce two new functions to be used in the \text{TA\_CreateEntryPoint} context. Listing 6 illustrates the signatures of \text{TEE\_RegisterCommand}, to register \text{(cmdId, paramTypes)} tuples for the \text{TA\_InvokeCommandEntryPoint} function, and \text{TEE\_RegisterOpenSession}, to register the paramTypes for the single command handler in the \text{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.

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.

---

\begin{verbatim}
1 \text{TEE\_Result TEE\_RegisterCommand(}
2 \quad \text{uint32\_t cmdId, uint32\_t paramTypes});
3 \text{TEE\_Result TEE\_RegisterOpenSession(uint32\_t paramTypes);} \end{verbatim}

---

\text{cmdId} 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 illustrates the inner workings of our mitigation. First, the TA registers a single command with cmdId 1 to have paramTypes \text{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., \text{entry\_invoke\_command} function), which eventually calls the TA’s \text{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 \text{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 \text{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 recurring 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 imple-

---

\text{cmdId} is the tuple of \text{cmdId} and paramTypes that needs to be fixed and enforced before any untrusted input is supplied to TAs.
mentation 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 bootloader 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 HR00119S0089-AMP-FP-034.

References


13 Appendix

<table>
<thead>
<tr>
<th>Vuln TA</th>
<th>Name</th>
<th>CVE</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_storage</td>
<td>Secure Storage</td>
<td>CVE-2016-8764</td>
<td>[54]</td>
</tr>
<tr>
<td>d76d338b1ac349e09f65f4efe179739da</td>
<td>Soter</td>
<td>None</td>
<td>[41]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-000000000046</td>
<td>MDLAP</td>
<td>SVE-2019-14867</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-000048444350</td>
<td>HDCP</td>
<td>SVE-2019-14850</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-00005854b50a</td>
<td>SKPM</td>
<td>SVE-2019-14892</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-0057645524ed</td>
<td>WVDRM</td>
<td>SVE-2019-14885</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-42495755542</td>
<td>EXT FR</td>
<td>SVE-2019-14847</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-46494c74502</td>
<td>FINGERPRINT</td>
<td>SVE-2019-14864</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-534545854652</td>
<td>SEC_FR</td>
<td>SVE-2019-14851</td>
<td>[43, 49]</td>
</tr>
<tr>
<td>00000000-0000-0000-0000-534545855345</td>
<td>SEM</td>
<td>SVE-2019-14891</td>
<td>[43, 49]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0-Day TA</th>
<th>Disclosure Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpeid</td>
<td>Confirmed</td>
</tr>
<tr>
<td>chnactiv</td>
<td>Confirmed</td>
</tr>
<tr>
<td>081100000000000000000000000000000000</td>
<td>CVE-2023-32835</td>
</tr>
<tr>
<td>080300000000000000000000000000000000</td>
<td>CVE-2023-32834</td>
</tr>
<tr>
<td>061400000000000000000000000000000000</td>
<td>CVE-2023-32848</td>
</tr>
<tr>
<td>090100000000000000000000000000000000</td>
<td>Confirmed</td>
</tr>
<tr>
<td>080200000000000000000000000000000000</td>
<td>Confirmed</td>
</tr>
<tr>
<td>080300000000000000000000000000000000</td>
<td>Confirmed</td>
</tr>
<tr>
<td>090300000000000000000000000000000000</td>
<td>Confirmed</td>
</tr>
<tr>
<td>98fb95bcb4bf24d2d674e3ae48690d7ea</td>
<td>Confirmed</td>
</tr>
<tr>
<td>abc270ea5c4458b43384a2fa2539e</td>
<td>Confirmed</td>
</tr>
<tr>
<td>077700000000000000000000000000000000</td>
<td>Confirmed</td>
</tr>
<tr>
<td>e97c270e-a5c4-4c58-bcd3384a2fa2539e</td>
<td>Confirmed</td>
</tr>
<tr>
<td>00000000-4d54-4b5f-4246-564448455643</td>
<td>CVE-2024-20078</td>
</tr>
<tr>
<td>00000000-4d54-4b5f-4246-564448455643</td>
<td>Confirmed</td>
</tr>
</tbody>
</table>

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