Modern iOS Security Features – A Deep Dive into SPTM, TXM, and Exclaves

Moritz Steffin and Jiska Classen

Based on the M.Sc. thesis by Moritz Steffin.

Abstract

The X is Not Unix (XNU) kernel is the basis of Apple's operating systems. Although labeled as a hybrid kernel, it is found to generally operate in a monolithic manner by defining a single privileged trust zone in which all system functionality resides. This has security implications, as a kernel compromise has immediate and significant effects on the entire system. Over the past few years, Apple has taken steps towards a more compartmentalized kernel architecture and a more microkernel-like design. Whilst initially limited to moving key sensitive components (page table manipulation, cryptographic functionality) to kernel-separated scopes, with the introduction of the Secure Page Table Monitor (SPTM) in 2023, this movement towards compartmentalization has intensified.

To date, there has been no scientific discussion of SPTM and related security mechanisms. Therefore, the understanding of the system and the underlying security mechanisms is minimal. In this paper, we provide a comprehensive analysis of new security mechanisms and their interplay, and create the first conclusive writeup considering all current mitigations.

SPTM is be the highest-privileged system component running in Guarded Level (GL) 2 using the Guarded Execution Feature (GXF). The GXF implements horizontal guarded exception levels in addition to the standard exception levels, and is used to protect sensitive system activity from unrestricted access by XNU.

SPTM acts as the sole authority regarding memory retyping. Our analysis reveals that, through SPTM domains based on frame retyping and memory mapping rule sets, SPTM introduces domains of trust into the system, effectively gapping different functionalities from one another. Gapped functionality includes, among others, the Trusted Execution Monitor (TXM), responsible for code signing and entitlement verification. We further demonstrate how this introduction lays the groundwork for the most recent security feature of Exclaves, and conduct an in-depth analysis of its communication mechanisms. We discover multifold ways of communication, most notably `xnuproxy` as a secure world request handler, and the Tightbeam Inter-Process Communication (IPC) framework. Finally, we provide a top-level overview of components running in GL and their interactions.

The architecture changes are found to increase system security, with key and sensitive components being moved out of XNU's direct reach. This also provides additional security guarantees in the event of a kernel compromise, which is no longer an immediate threat at the highest trust level.

Contents

Acronyms

ANE Apple Neural Engine. 62, 99, 107
AP Access Permission. 9, 11
APRR Access Permissions Remapping Register. 10, 11

CTID Compact Thread ID. 78

DCP Display Coprocessor. 99
DMG Apple Disk Image. 54, 98
DYLD Dynamic Linker. 63

EIC Exclave Indicator Controller. 98
EL Exception Level. 21, 30, 34
ELR Exception Link Register. 9, 11
ESR Exception Syndrome Register. 9, 11

FPR Fast Permission Restrictions. 10, 11
FTE Frame Table Entry. 38–40

GL Guarded Level. iii, 11, 17, 21, 30, 49, 64
GXF Guarded Execution Feature. iii, 7, 11, 16, 17, 19, 20, 33–35, 47, 104

HCR_EL2 Hypervisor Configuration Register EL2. 35
HVC Hypervisor Call. 18, 19, 21, 24, 30, 34, 35

IOMMU Input/Output Memory Management Unit. 29
IPC Inter-Process Communication. iii, 6, 7, 18, 54, 56, 60, 65, 67–71, 74, 75, 78, 79, 96, 107

NSA National Security Agency. 15

PAPT Physical Aperture Table. 41
PMAP Physical Map. 42
POSIX Portable Operating System Interface. 61, 63
PPL Page Protection Layer. 6, 10, 11, 17, 103
PTE Page Table Entry. 9, 11, 42, 44, 45
PXN Privileged Execute Never. 9, 11

SCID Scheduling Context ID. 78, 79
SDK Software Development Kit. 25
SIL Secure Indicator Light. 64, 98, 99

SK Secure Kernel. 18, 24, 25, 29, 31, 33, 38, 44, 47–49, 52, 54, 66, 78, 100, 107

SoC System-on-a-Chip. 6, 10

SPRR Shadow Permissions Remapping Register. 11, 12, 17, 20, 41, 45, 46, 104

SPSR Saved Program Status Register. 36, 37

SPTM Secure Page Table Monitor. iii, 11, 14, 17–19, 21–31, 33–35, 37–47, 49, 57, 71, 78, 100, 102, 103, 105–107

SRD Security Research Device. 14, 15, 64

SVC Supervisor Call. 8, 18, 19, 21, 24, 30, 34–36, 49, 51, 54, 100, 106

TGE Trap General Exception. 35

TPIDR Software Thread ID Register. 11

TXM Trusted Execution Monitor. iii, 14, 18, 19, 21, 25, 26, 29, 31, 33–35, 37, 38, 100, 103, 107

UXN Unprivileged Execute Never. 9, 11

VBAR Vector Base Address Register. 9, 11, 20

XNU X is Not Unix. iii, 6, 15, 17, 19, 21, 43, 46, 55, 56, 64, 70, 75, 79, 83, 86, 87, 92, 93, 100, 105

XPC Cross-Process Communication. 97

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Motivation

Apple's XNU kernel builds the base of the Darwin operating system. Upon this operating system, a multitude of Apple proprietary operating systems, such as macOS and iOS, are built. According to Apple, the XNU kernel is a hybrid kernel [34]. It combines the *Mach microkernel*, built at Carnegie Mellon University, with the monolithic FreeBSD kernel [38].

Operating systems based on monolithic kernels run "every basic system service" in the privileged kernel space. This includes services such as process management, interrupt handling, drivers, and IPC. In contrast to that, microkernel-based operating systems provide a very basic set of functionality in the kernel space, including "basic process communication and I/O control", whilst placing other system services in user space [44]. While microkernels are generally considered superior in terms of system security due to their reduced attack surface compared to components running at the highest privilege levels, they are typically regarded as inferior in terms of performance.

Despite its labeling as a hybrid kernel, the implementation of the XNU kernel is monolithic mainly, as it "places all system functions within the same privileged scope" [21]. This implementation comes with the known security flaws of monolithic kernels. Especially the lack of kernel address space separation poses a non-negligible security risk. With Apple's chosen implementation, faults and weaknesses in any system call or driver can be exploited to "maliciously manipulate kernel code", posing a risk of a full and privileged kernel compromise, as the kernel is the most privileged part of the system [39].

With the release of the SPTM and the TXM for A15/M2 or newer System-on-a-Chip (SoC) in 2023, Apple is addressing these issues by beginning to compartmentalize its kernel into different privilege spaces. This aims to "protect page tables for both user and kernel processes against modifications, even when the attackers have kernel write capabilities [...]". As advertised on the Apple Support pages, these systems are intended to replace the previous Page Protection Layer (PPL), which we will investigate. They aim to "provide[] a smaller attack surface that does not rely on trust of the kernel" [45]. This was further advanced with the introduction of Exclaves, a security feature designed to isolate sensitive operations away from XNU.

Publicly available information regarding the actual implementation and implications of these features is scarce. To date, Apple has not issued any official releases on this matter, apart from the previously cited minor publications in the context of operating system integrity guides. Therefore, to infer a potential change in security and the underlying threat model addressed, a deep analysis of the underlying architecture is of high interest. In doing so, we aim to provide an introduction to more intensive research into modern iOS security and memory system mechanisms.

1.2 Research Goals & Contributions

Our analysis aims to provide an initial understanding of the architectural design of the SPTM-based system and memory security. We shed light on the underlying security assumptions and their impact on the security of real-world systems. Public analysis of modern iOS security features, such as SPTM, TXM, and Exclaves, and their underlying

mechanisms, is minimal. Most sources focus on individual aspects or are limited in their applicability [25, 26, 27]. While working on this topic, more detailed blog posts were published [21, 43, 42], but various open questions remain. Furthermore, official information from Apple is extremely limited. Our contributions are as follows:

- Providing the most comprehensive analysis to date of components running in GXF, their interaction with each other, and interaction with XNU, in light of the recent restructuring of GXF components with the release of Exclaves.
- Reverse engineering of SPTM with a focus on its interaction with its clients and the discovery of its underlying rule set regarding memory frame retyping and memory mapping, as well as the derived security implications, despite the lack of any prior official information regarding its inner workings.
- First comprehensive public discussion of Exclave communication mechanisms with a focus on Tightbeam and xnuproxy. Discovery of Tightbeam as an Exclaves secure world IPC framework.
- To our knowledge, we have performed the first practical experiment in directly interacting with conclaves from XNU user space.

1.3 Outline

Our work aims to provide a view into individual components and their interplay in the realization of memory and system security. We will begin by giving a brief top-down overview of the system in question (see Chapter 4) and then proceed to a more detailed analysis of its specific components. Our analysis of SPTM will be the core part of this work (see Chapter 5), with a focus on the underlying security mechanisms and implications. Furthermore, we provide insights into Exclaves, including their underlying memory safety assumptions, and focus on communication mechanisms (see Chapter 7). As Exclaves run in the GXF, we examine the Secure Kernel (see Chapter 6), a central component of GXF, and provide a brief overview of TXM. Our focus in this analysis will be the cross-component communication and memory security based on SPTM.

2 Fundamentals and Background

2.1 AArch64 Fundamentals

The chips in use in Apple's hardware are based on the AArch64 architecture by ARM. While Apple employs a variety of proprietary, Apple-specific customizations, the architectural fundamentals still apply and build the base for the respective firmware. As the security mitigations examined in this paper primarily focus on granular privilege scoping and memory access privileges, two key components of the architecture are AArch64 exception levels and AArch64 memory addressing.

2.1.1 AArch64 Exception Levels & Exceptions

Exception levels are the AArch64 realization of privilege levels. They are usually referred to as `EL_X`, whereby `X` is the exception level in question. Higher exception levels are associated with higher privileges. The AArch64 architecture has four native exception levels, as shown in Figure 2.1.

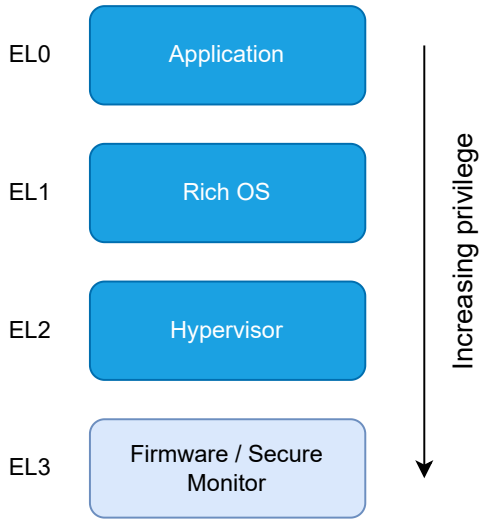


Figure 2.1: Exception levels on AArch64 with the common usage model applied, adapted from [8].

The usage of the above exception levels is not strictly defined by the architecture, but rather by the common usage model. Switching between privilege levels typically occurs when taking an exception or returning from an exception. However, it may also change during the debug state, upon exiting the debug state, and upon processor reset [8].

An exception in the sense of the ARM architecture model is "any condition that requires the core to halt normal execution and instead execute a dedicated software routine known as an exception handler [...]" [10]. An example of this is an EL0 application making Supervisor Calls (SVCs) to invoke system functionality at a higher privilege level [15].

The architecture utilizes a variety of key system registers to facilitate exception handling. These include a register holding information on the reason for an exception (Exception Syndrome Register (ESR)), a register holding the address of the instruction causing the exception (Exception Link Register (ELR)), and a register that sets the exception base address for exception handling in a specific exception level (Vector Base Address Register (VBAR)) [17]. We will have a more in-depth look at these specific registers and their usage in Chapter 5.

2.1.2 AArch64 Memory Access

Memory access in the AArch64 architecture is realized via a virtual address space. Instead of directly operating on physical memory addresses, applications are provided with virtual addresses by the operating system. This allows for easy sandboxing by giving each application own address space, or for abstraction on the underlying hardware [19].

The operating system performs the mapping between physical and virtual memory through so-called translation tables. At its essence, they hold entries for every virtual address, with a mapping to the corresponding physical address, along with various flags and attributes. In practice, this lookup table system is typically multilevel, with indirection from translation tables to further higher-level translation tables. However, the key concept of memory mapping remains the same [12]. Additionally, mapping is usually not done on individual memory addresses, but instead on predefined-sized memory chunks. A memory chunk in the *virtual* address space is referred to as a memory *page*, while a memory chunk in the *physical* address space is referred to as a memory *frame*.

Figure 2.2 shows a simplified level one translation table block entry layout for the ARMv8-A architecture (on which the AArch64 execution state is based). Whilst different kinds of entries and entries on other translation table levels may differ, the general concept remains the same.

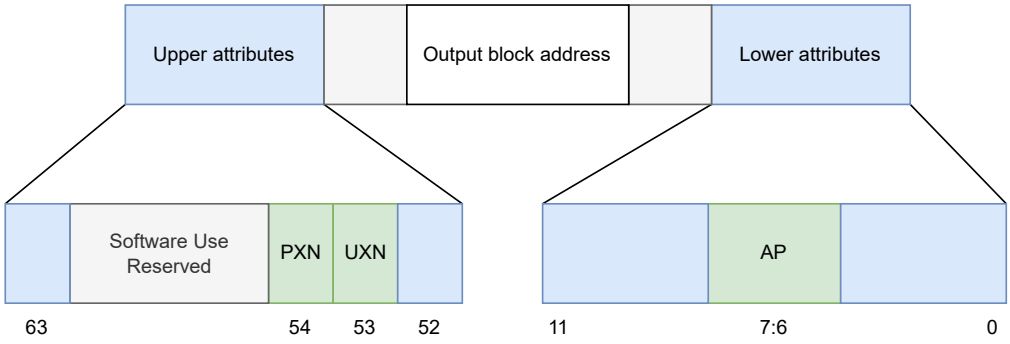


Figure 2.2: Simplified ARMv8-A translation table block entry, adapted [9].

The block entry displays the architecture's default memory access permission mechanism, as represented by the so-called Access Permission (AP) bits, at bit positions 6 and 7. They encode the memory access permission, with the interpretation being dependent on the context of privileged (EL1+) and unprivileged (EL0) access attempt. The underlying encoding is illustrated in Table 2.1.

As the AP bits only encode access (i.e., read/write) permissions, an additional mechanism is required for execution permissions. This is implemented via the Unprivileged Execute Never (UXN) and Privileged Execute Never (PXN) bits at bit position 54 and 53, respectively. If the relevant bit is set, the memory mapped by the specific Page Table

Access Permission Bits	Unprivileged (EL0)	Privileged (EL1/2/3)
00	No access	Read/Write
01	Read/Write	Read/Write
10	No access	Read-only
11	Read-only	Read-only

Table 2.1: Access permissions for different exception levels [13].

Entry (PTE) is marked as non-executable from both the unprivileged/privileged exception levels. Note that there are additional mechanisms that modify how the exact permission is interpreted, depending on factors such as the processor state at the time of access [13].

2.2 Exception Levels Usage for iOS

Apple operating systems are built upon the ARMv8 architecture. Therefore, the implementation of privilege levels is done via ARMv8 exception levels. In Apple's interpretation of the common usage model, user-space code (i.e., applications, most daemons) runs on EL0. Apple provides no reliable information as to which exception level the kernel uses, but we find a multitude of both EL1 and EL2 register accesses in the kernel binary. This indicates that Apple deviates from the standard of running a hypervisor in EL2, but instead runs the kernel there. This is nothing out of the ordinary per se, and can be achieved with techniques like ARM64 Virtualization Host Extensions [40]. Since 2019, iOS abandoned the concept of running a secure monitor in EL3 [54].

2.3 Related Mitigations

The introduction of Exclaves and the deployment of SPTM is not the first novel approach towards compartmentalizing the XNU kernel, but rather the latest release in a line of multiple previous mitigations released over the past few years, aimed at enhancing kernel security. To provide a basis for the most recent mitigations, this section provides a brief background on a selection of these.

2.3.1 Fast Permission Restrictions

Fast Permission Restrictions (FPR) are a hardware primitive used for the fast and efficient remapping of memory permissions, without the need for computationally heavy page table walks. For example, it can help to ensure secure just-in-time compilation as it can quickly switch between writable and executable memory. This fundamental ability is the core capability for other mitigations that have been released throughout the past years.

FPR are realized through special APRR registers [45]. They have been introduced with the 2017 Apple A11 Bionic SoC. Whilst not officially confirmed, APRR is assumed to denote Access Permissions Remapping Register (APRR). As this exact implementation is no longer in use with current devices, we consider it out of scope. It is, however, comparable to the implementation of the SPRR registers, which is looked at in Section 2.3.4.

2.3.2 Page Protection Layer

Introduced in 2017, Apple designed PPL to deny user-space code modification after successful code signature verification. PPL was introduced as the new sole mean to alter

specific protected pages, especially those containing page tables. PPL was realized by employing the previously mentioned FPR and, more specifically, the proprietary APRR register [45]. PPL offers a specific trampoline code (`__PPLTRAMP` segment), which, after invoking it, employs the APRR register to efficiently remap large amounts of page table permissions, and vice versa on exit [35]. The actual protection is therefore achieved by mapping sensitive memory (in this case, mainly page tables) and the PPL code itself as `r--` (read-only) to the kernel. The PPL trampoline uses the APRR register to flip the permissions of the page tables to `rw-` (read-write) and the actual PPL code as `r-x` (read-execute). This PPL-trampoline is the only entry point to PPL, and therefore significantly reduces the attack surface with respect to page table manipulation [40]. SPTM has replaced the PPL mitigation on A15 / M2 or newer SoCs [45].

2.3.3 Guarded Execution Feature

The GXF, presumably released in 2021, introduces lateral exception levels next to the architecture's default exception levels. These levels are called Guarded Levels (GLs). Entering these GLs is achieved via the Apple-proprietary `GENTER` instruction (opcode: `0x00201420`), whilst exiting is done via `GEXIT` respectively (opcode: `0x00201420`). Comparable to standard AArch64 exception levels, the GLs have relevant system registers (such as Software Thread ID Register (TPIDR), VBAR, ESR, and ELR), those being another Apple-proprietary architectural addition [28]. The lateral levels introduce a different set of permissions compared to default exception levels, allowing for a more granular distribution of privilege. This is implemented via the so-called Shadow Permissions Remapping Registers (SPRRs).

2.3.4 Shadow Permission Remap Register

The SPRRs are the tool used to implement the GXF. They reimplement the usage of the Access Permission (AP) bits and the UXN/PXN bits in page table entries looked at previously by adding a layer of indirection to them (initially introduced via APRR). Instead of directly encoding the memory access/execute permissions into the relevant bitfields, they are used to generate an index for a lookup into a system register [40, 28]. The bitwise SPRR index calculation based on the permission fields of a Page Table Entry (PTE) is shown below:

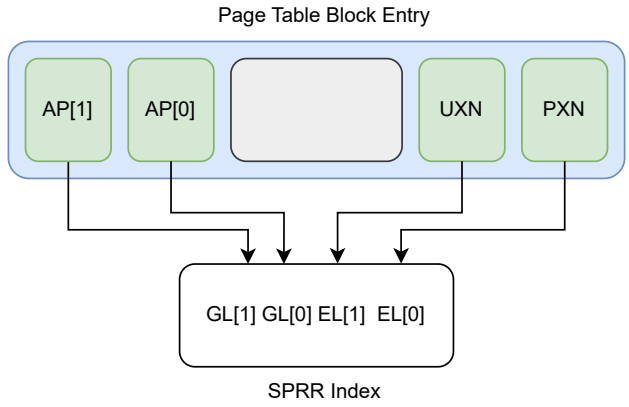


Figure 2.3: SPRR index calculation based on the previously seen arguments AP, UXN, and PXN in a page table block entry, adapted [28].

The retrieved index is then used to access privilege level-specific system registers, in which the actual permissions are encoded. The concept of this index retrieval is shown in Figure 2.4.

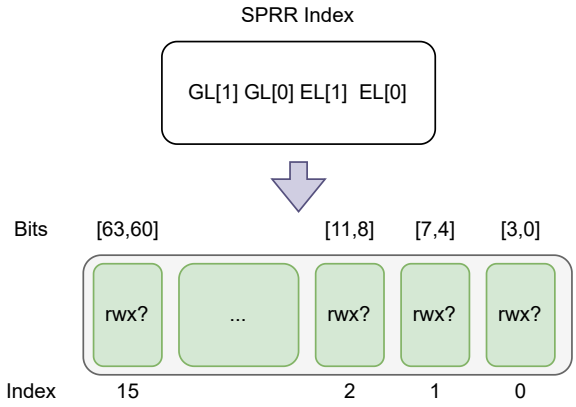


Figure 2.4: Permission lookup scheme in SPRR system register based on the SPRR index.

In 2021, Sven Peter reverse engineered the permissions mapped behind the usage of SPRR on an M1 [40]. The following table is based directly on his findings. The exact permissions mapped to each index have likely changed since then, but we deem the table informative enough to understand the general workings of SPRR permissions. The table also includes found index usage in Peters's reverse engineering.

Index	SPRR EL0	SPRR EL2	SPRR GL2	Usage
1	`---`	`r--`	`rw-`	Page Tables
3	`---`	`rw-`	`rw-`	Kernel Data
5	`rw-`	`r-x`	`r--`	Userland MAP_JIT
7	`rw-`	`rw-`	`rw-`	Userland Data
8	`---`	`r--`	`r-x`	PPL Code
10	`---`	`r-x`	`r-x`	Kernel Code
11	`---`	`r--`	`r--`	Kernel Read-only Data
13	`r-x`	`r--`	`---`	Userland Code
15	`r--`	`r--`	`---`	Userland Read-only Data

Table 2.2: SPRR index to permissions mapping and usage [40].

It is noteworthy that at the time of initial release, all remaining indices showed no access permissions on any privilege level. This indicates that there was still room for a more granular implementation of fast-mapped permissions.

Implications for Security As shown in Table 2.2, the SPRR-based permissions significantly restrict the possibility of altering page tables. Whilst read access is still possible from EL2, page table writes are completely encapsulated in GL2. Page tables, and as such, a highly vulnerable part of the operating system, are protected even from privileged kernel code, requiring the use of very specific `GENTER/GEXIT` functionality to write access them.

2.4 Previous Work

The Apple security mechanisms analyzed in this work have, to this date, not yet been analyzed in any scientific paper. This is partially due to their recency, but also a result of Apple's lack of public communication regarding their releases and a near-complete lack of official sources on their inner workings. There are few public sources on SPTM, TXM, and Exclaves.

Dataflow Forensics has provided a fairly comprehensive write-up about SPTM, TXM, and their XNU interaction in the form of three blog posts spanning from August 2023 to February 2025. The blog posts provide an entry point into understanding SPTM, and especially shed light on communication between XNU and SPTM, as well as SPTM and TXM. As SPTM and components running in Apple's GXF are subject to constant change, especially since they are relatively new, the posts do not, in parts, reflect the current state of things. Nonetheless, we build our analysis of SPTM upon them and the concepts they display. They will be referenced repeatedly throughout this work for core aspects of SPTM understanding.

On the question of Apple Exclaves, the only original public discussion is found in three key blog posts by Random Augustine [21, 43, 42]. They discuss the general concept of Apple Exclaves and provide a high-level analysis, primarily based on XNU public sources. We will reference this work in our analysis of Exclaves, as it provides a fairly in-depth look into Exclave thread scheduling, a concept we have excluded from our analysis.

3 Setup

The following analysis has primarily been conducted on specific hardware and firmware specifications. As the topic of SPTM, TXM, and Exclaves is a highly recent one, these implementations are naturally subject to change. Therefore, the findings presented in this paper may no longer apply to newer hardware and firmware. Nonetheless, in favor of practicability, we have chosen and decided on specific versions and have not taken newer releases into account for most of this paper. This poses no issue, as any analysis and reverse engineering of still actively developed software, firmware, and hardware is always just a momentary snapshot of the current state. The following section will shortly list the used hardware and firmware specifications.

3.1 Firmware

Regarding firmware, the analysis was primarily conducted on the iOS 18.4 (22E240) firmware for the iPhone 16. The firmware was obtained via `https://appledb.dev`. The release correlates with the kernel version *xnu-11417.102.9*.

Significant parts of the following analysis are based on Apple's open source publications of kernel sources [34]. The primary open source release used was *xnu-10063* [32], due to its availability at the time of our research. All references to XNU open-source code without an explicit version specification shall refer to this version. We furthermore declare that all path specifications provided throughout this work, if not specified otherwise, shall refer to said XNU open source code.

3.2 Hardware

Regarding the dynamic analysis of the new security mitigations, multiple hardware devices were utilized.

1. iPad Pro 11-inch (M4) running iPadOS 17.5.1, used for initial log analysis via the macOS Console app.
2. iPhone 16 Security Research Device (SRD) running iOS 18.5, used for dynamic reverse engineering on recent iOS versions.
3. iPhone 8 running iOS Version 16.7.10, jailbroken via the palera1n jailbreak version v2.1-beta [50]. This device was used for dynamic reverse engineering in the context of security analysis comparison on the *Recording Indicator/Secure Indicator Light*.

Security Research Device

The *Apple Security Research Device Program*[1] allows specific individuals with a strong and proven background in security research on Apple and other modern operating systems to obtain so-called SRD from Apple. These devices "allow[ ] you to perform iOS security research without having to bypass its security features" [33]. Such a research device

[1] Apple Security Research Device Program: `https://security.apple.com/research-device/`, last accessed: 14.09.2025

enables the analysis of even the most recent firmware versions without waiting for the corresponding jailbreaks to be released. The capabilities on these devices are extensive, as they support shell access, allow for setting individual entitlements, and even support kernel customization.

The SRD used for analysis during this paper was provided by Apple to Dr. Jiska Classen. In compliance with Apple's Security Research Device guidelines, all analysis on the device itself was performed by Dr. Jiska Classen, the authorized Apple user.

3.3 Tooling

As this paper was mainly focused on static reverse engineering, we employed a variety of such tools.

Ghidra Ghidra is a software reverse engineering framework developed by the National Security Agency (NSA) [1]. It has been the primary tool used in our analysis.

ipsw A command-line research tool for iOS and macOS, and has been used in our analysis to dissect Apple IPSW files [23].

disarm A command-line binary analyzer we additionally employed for reverse engineering [36].

symbolicator A tool used for symbolication of XNU binaries [24].

lldb The debugger from the LLVM project, used for dynamic tracing of calls on the SRD [37].

3.4 Binary Naming Convention

To allow for easier reference to binaries analyzed in this work, we will define the following naming convention for binaries at paths in the firmware:

- SPTM binary:
  `22E240__iPhone17,3/Firmware/sptm.t8140.release`
- TXM binary:
  `22E240__iPhone17,3/Firmware/Firmware/txm.iphoneos.release`
- Exclavecore binary:
  `22E240__iPhone17,3/Firmware/image4/exclavecore_bundle.t8140.RELEASE`
- Secure Kernel binary:
  `22E240__iPhone17,3/Firmware/image4/SYSTEM/kernel`
- XNU kernelcache binary:
  `22E240__iPhone17,3/kernelcache.release.iPhone17,3`

3.5 Symbol Enumeration

During our reverse engineering for this work, we symbolicated a variety of different functions and structures in all the binaries we examined. We include a table with symbol-to-address mappings for the various binaries analyzed in Section A.1, which ensures reproducibility of our analysis.

3.6 Apple-Proprietary Registers

For the realization of features like the GXF, Apple uses its own custom and proprietary registers. As these specifications are usually not officially released, some reverse engineering tools may not yet have incorporated these. To facilitate an understanding of the underlying functionality, we have utilized releases of these Apple-specific system registers from previous reverse engineering work. As these are unofficial, there is no claim to correctness and completeness. These sources include a recent leak as well as Asahi Linux's tools [20, 41] For the sake of clarity, we will use the definitions of Asahi Linux's `m1n1` tool, as they form the basis of our reverse engineering work.

4 Overview on Architectural Design

This chapter provides a brief overview of the system's architectural design, as reverse-engineered by us. We will shortly introduce the underlying usage of GXF in Apple operating systems. In doing so, we provide the fundamentals for the upcoming in-depth analysis of system components and their interactions. An overview of the system, our assumptions, and findings based on this analysis are shown Figure 4.1.

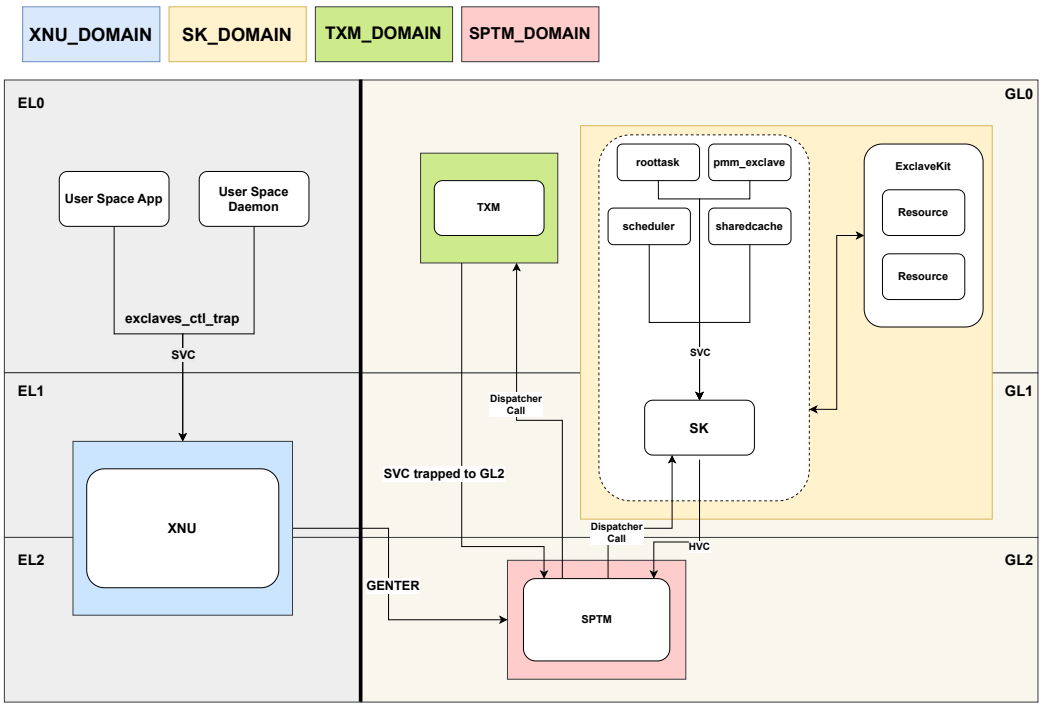


Figure 4.1: Overview of the system analyzed by us, with focus on communication paths between individual components across ELs and GLs.

System Layout

The novel system is based on the Guarded Execution Feature (GXF) we have previously introduced in Section 2.3.3. We find that all components (except the XNU kernel) analyzed by us are running in Guarded Levels (GLs), for which we know different access and execution permissions are realized via SPRR (see Section 2.3.4).

The new GXF central management and dispatching component is Secure Page Table Monitor (SPTM). XNU can call into SPTM via the proprietary `GENTER` instruction, and based on a provided dispatch target, SPTM either handles the request itself or dispatches the call into other guarded-level components. SPTM replaces the PPL in performing page table manipulation. It has further advanced its responsibilities regarding memory safety,

introducing the concept of SPTM domains. These domains act as a source of trust, with limited memory mapping interaction across domains.

Regarding other protected-level components, Trusted Execution Monitor (TXM) is running in GL0 and is responsible for, among other things, code signing and entitlement validation. It can call into SPTM via SVC instructions.

With Exclaves, the guarded levels are extended with a complex and extensive system. Exclaves are found to be access-scoped groupings of sensitive resources, which require special privileges to call into, and reduce XNU's ability to interact with security-relevant operations. The Exclave ecosystem introduced the Secure Kernel (SK), which handles SPTM requests towards Exclaves and serves as a GL1 microkernel. The SK can call into SPTM via Hypervisor Call (HVC) instructions. Furthermore, a variety of Exclave-related GL0 binaries have been introduced, which appear to handle, among other things, Exclave scheduling and calling. We discover Tightbeam as a newly introduced IPC mechanism for calling into Exclaves.

5 Secure Page Table Monitor

5.1 SPTM Fundamentals

SPTM is a GXF component running in GL2[2]. GL2 is the highest guarded level observed so far, and therefore the most privileged one across the entire system. SPTM is the sole component detected so far running in that privilege level, and is therefore the most privileged component. This puts it in a unique position, making it responsible for the most security-relevant operations.

We will demonstrate that SPTM is invoked from XNU via `GENTER`, with this call entering the GXF. TXM, running in GL0, may call directly into SPTM via SVCs. This is achieved through a special hypervisor configuration for SPTM trapping exceptions, targeting GL1 to GL2. GL1 components (the Secure Kernel) may call into SPTM directly via HVCs.

We find SPTM to be the key component responsible for performing memory mappings, which it restricts based on a specific ruleset. These restrictions are demonstrated through SPTM frame types and SPTM domains, which scope components into privilege spaces concerning memory mapping. This compartmentalization within privileged kernel code appears to be the first step towards a more microkernel-oriented XNU architecture, which is further supported by our finding of seL4 microkernel components used to realize Exclaves (see Chapter 7).

5.2 SPTM Setup

We know SPTM to be running in GL2 and therefore the GXF. As previously looked at, GXF is entered via the `GENTER` instruction. Among other GXF configurations, the GXF entry point, that is, the address to be jumped to when entering GXF, is defined via a system register called `GXF_ENTRY_ELx` [40]. Looking for these registers in SPTM reveals two GXF setup functions. The two functions are shown in Listing 5.1 and 5.2.

```
1 void gxf_setup_early(void)
2
3 {
4   GXF_CONFIG_EL1 = 1;
5   GXF_PABENTRY_EL1 = 0xffffffff027097934;
6   GXF_PABENTRY_EL1 = 0xffffffff02708b88c;
7   InstructionSynchronizationBarrier();
8   GENTER();
9 }
```

Listing 5.1: `gxf_setup_early` function in SPTM performing GXF setup after initial SPTM entry. The source code was disassembled by Ghidra.

[2]We know this from GL2 register calls.

```
undefined8 gxf_setup_late(void)

{
  undefined8 uVar1;

  spsel = 1;
  spsel = 0;

  GXF_PABENTRY_EL1 = 0xfffffff027097934);
  GXF_ENTRY_EL1 = 0xfffffff02708053c;
  VBAR_GL1 = 0xfffffff027084000;

  InstructionSynchronizationBarrier();

  uVar1 = 0x2f;
  if (DAT_fffffff027079618 == '\0') {
    GXF_CONFIG_EL12 = 0;
    GXF_ENTRY_EL12 = 0;
    GXF_PABENTRY_EL12 = 0;
    uVar1 = 0x6f;
  }
  GXF_CONFIG_EL1 = uVar;
  InstructionSynchronizationBarrier();
  return uVar1;
}
```

Listing 5.2: `gxf_setup_late` function in SPTM performing GXF setup, not part of the initial SPTM entry process. The source code was disassembled by Ghidra.

The `GXF_ENTRY` registers store the GXF entry vector for the specified exception level, and `GXF_PABENTRY` registers store the respective GXF abort vector [20]. We assume that `GXF_CONFIG` stores general GXF configuration for the specific exception level, but no detailed information is available on this register.

GXF Entry Point – Early Setup The core setup in the `gxf_setup_early` function sets the `GXF_entry` register. The address stored is the entry vector into GXF; therefore, the address that is jumped to on `GENTER` invocations. The address points to a function we named `GXF_early_entry`, which performs context setting. Most notably, it sets the EL1 SPRR kernel permission configuration register `SPRR_PPERM_EL1`, which encodes access permissions for EL1 kernel components. This might ensure specific permissions for kernel components during guarded-level execution. It further sets the EL1 SPRR configuration register (`SPRR_CONFIG_EL1`) to `0xff`, enabling SPRR, and locking down its configuration and both user and kernel space permissions, preventing further alterations [40, 20]. The key setup part regarding further request handling is setting the GL1 VBAR, which we can assume works similarly to the EL1 equivalent known to point to a base address, offset from which exception handlers are registered [18, 16].

Looking at exception handlers at the defined offsets [16], we find a pattern of reading exception information based on the calling context, and storing it in memory structures. There appears to be no further request handling performed. We assume this to be a very early request handling setup, which handles incoming requests by storing them, but does

not actually act on them. This is in line with this GXF entry point being set nearly directly after the SPTM `entry` point.

GXF Entry Point – Late Setup As the early setup GXF entry point did not reveal much regarding actual SPTM request handling, we assume that `GXF_late_setup` performs the actual request handling setup. Different from the previous `GXF_setup_early` function, it only sets the `GXF_PABENTRY` and `GXF_ENTRY` registers. It additionally directly sets the `VBAR_GL1` register, storing an address based on which we find a variety of actual exception handlers. We will examine the request handler in more depth in Section 5.4.3. The address stored in the `GXF_ENTRY_EL1` register points to a function we denote as `GXF_entry_point`, which handles `GENTER` invocations and will be examined in Section 5.4.2. We assume `gxf_setup_late` to be executed in later setup stages of SPTM, when SPTM is actually prepared to begin handling requests, and can therefore set up the required handlers.

5.3 Calling into SPTM

As SPTM, in its essence, provides services to clients. These clients (among them XNU and TXM) must be able to invoke such services. As previously determined, XNU interacts with the GXF and SPTM via the `GENTER` instruction, as it has to switch from the highest Exception Level (EL) to a Guarded Level (GL). Note that this is not the case for other SPTM clients already running in a GL, which instead use an SVC or HVC. When XNU calls into SPTM, `GENTER` is parameterized via the `x16` register. An exemplary code fragment that shows this parameterization and call into SPTM is shown below.

```
1 ff00ab30080 10 00 e0 f2     movk        x16,#0x0, LSL #48
2 ff00ab30084 10 00 c0 f2     movk        x16,#0x0, LSL #32
3 ff00ab30088 10 00 a0 f2     movk        x16,#0x0, LSL #16
4 ff00ab3008c f0 01 80 f2     movk        x16,#0xf
5 ff00ab30090 20 14 20 00     GENTER
```

Listing 5.3: Parameterized `GENTER` call from XNU, with `x16` register setup followed by `GENTER` invocation.

This is a recurring pattern that can be found repeatedly in SPTM clients. Previous research indicates that the parameter in the `x16` register denotes a so-called "subsystem" together with a calling code [26]. This assumption does not hold entirely and is instead corrected by a later release of SPTM headers directly from Apple (see Section 5.3.4). However, it remains a valid working hypothesis.

The following sections will list calls to SPTM from various clients and examine their characteristics. An overview of all detected XNU calls into SPTM is shown in Figure 5.1.

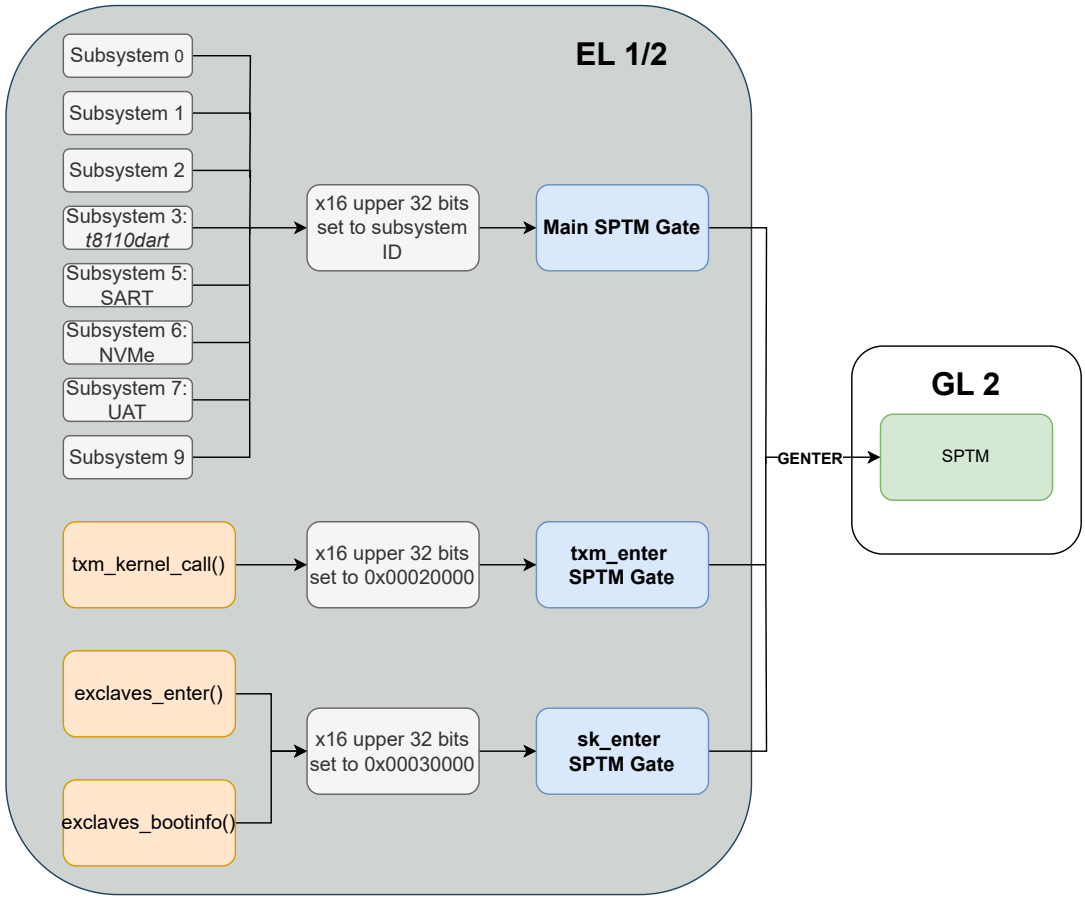


Figure 5.1: Depiction of XNU call structure into SPTM – SPTM is called through three specific gates. The main SPTM gate is used by what appears to be a multitude of different subsystems with calls parameterized via the subsystem ID. Two special SPTM gates are only invoked from individual functions and employ different `x16` parameterization schemes.

5.3.1 Calls from XNU

XNU makes various calls to different subsystems for SPTM. It does so by invoking `GENTER` from different "SPTM gates" depending on the calling context, which we will look at and list the relevant parameterized calls. To do this, XNU uses multiple different functions to invoke `GENTER` in different contexts.

Main SPTM Gate We find a function invoking `GENTER` in XNU, which we labeled `GENTER_main_gate`. Based on its frequent usage (77 direct calls), we infer that this is the **main SPTM caller**. The function calls a function that we consider to be a minor `GENTER` setup function (`genter_setup`), which verifies that the current privilege level is not yet guarded. After this, `GENTER` is called directly.

However, we are more interested in the calling functions, as we find that these set the `x16` parameter via the scheme listed in Listing 5.3. Enumerating the `x16` parameters used shows the following calling values from XNU forwarded to the aforementioned default SPTM entry point depicted in Table 5.1. We are furthermore able to partially map

specific subsystems to names by matching symbolized functions that call into the specific subsystems SPTM calls.

Subsystem ID	Subsystem Name	#Calls	Call Values	Invoking Functions - Excerpt
0	-	22	0x0-0xe, 0x10-0x15, 0x21	
3	t8110dart	17	0x300000000 - 0x300000010	`_t8110dart_iovmfree`, `table_t8110dart_map`, `_t8110dart_ioctl`
5	SART	3	0x500000000 - 0x500000002	`_sart_map`, `_sart_unmap`
6	NVME	8	0x600000000 - 0x600000007	`_nvme_ioctl`, `_nvme_init`
7	UAT	13	0x700000000 - 0x70000000c	`_uat_ioctl`, `_uat_get_ptep`, `_uat_init`
9	CPUTRACE	13	0x900000000 - 0x90000000c	

Table 5.1: Call values for `GENTER` calls into SPTM parameterized via `x16`. Subsystems are named, if possible, by the given context and calling functions.

These subsystem name assignments can be validated using the approach of Dataflow Forensics [26] by examining the SPTM binary. We find structures that we shall denote as subsystem descriptors. They are identifiable by the usage of the subsystem name string. Their structure is shown below in Figure 5.2.

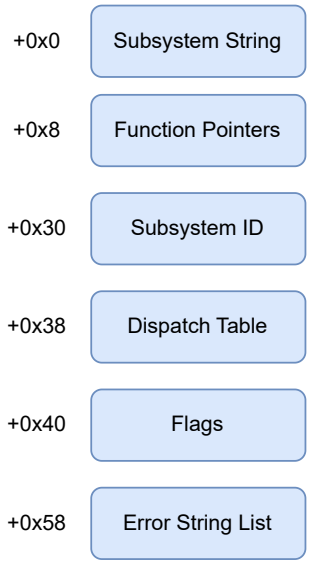


Figure 5.2: Layout of a subsystem descriptor as found in the SPTM binary.

The descriptor contains the subsystem name, a number of function pointers, the subsystem ID, a pointer to a list of functions that we assume to be a dispatch table, various options and flags, and a list of error strings. We can confirm our previously identified subsystems through this, and find a fourth subsystem called `CPUTRACE`, which we find to be subsystem number 9. The subsystem descriptors are detectable via their use of the name string. We will show their actual use in Section 5.3.5.

txm_enter – SPTM Gate We find another GENTER-calling function in XNU. Unlike the main SPTM gate, parameterization occurs directly before the call, not in another calling function. This specific GENTER call is dynamically parameterized. The upper 32 bits are statically set to 0x00020000, while the lower 32 bits are dynamically set based on a call parameter. We assume this SPTM gate to be txm_genter. We can infer this based on the only calling function to it, which we, in turn, assume to be txm_kernel_call, based on debug string usage in bsd/kern/txm.c in the XNU open-source code. Based on the GENTER call in the binary and the usage in the aforementioned file, we see the lower 32 bits of the x16 register are parameterized via a selector parameter, which we assume indicates a specific function to invoke. We will enumerate these function selectors and further look into them in Chapter 8.

sk_enter – SPTM Gate We find a third GENTER call in XNU. Comparable to the txm_enter call, parameterization also occurs right before the GENTER call and is also dynamic based on an argument regarding the lower 32 bits of x16. The upper bits of x16 are statically set to 0x00030000. There are two invocations of this SPTM gate found in the XNU binary. Based on the debug string usage in osfmk/kern/exclaves.c and comparison of control flow, we assume the invoking functions to be exclaves_enter and exclaves_bootinfo respectively. Using this knowledge, we can deduce that this specific SPTM gate corresponds to the sk_enter function. We can further infer the potential values with which sk_enter can be called. The endpoint, which is stored in the lower 32 bits of x16, is either RINGGATE_EP_ENTER or RINGGATE_EP_INFO, which we can assume to be values 0 and 1, respectively. In summary, we identify the following SPTM calls from this gate:

Secure Kernel Call #0 x16 value 0x3000000000000, corresponds to exclaves_enter

Secure Kernel Call #1 x16 value 0x3000000000001, corresponds to exclaves_bootinfo

5.3.2 Calls from the Secure Kernel

As already mentioned, the GENTER call serves as a gate into the SPTM via the GXF. As will be shown later, the Secure Kernel (SK) already runs in guarded mode. As it is even running at a privileged guarded level (GL1), it does not invoke SPTM via SVCs, but instead via HVCs. Nonetheless, we can still find similar x16 parameterization. This section will only briefly list SPTM calls from the SK without further delving into their workings. We will look at these in Chapter 6.

The secure kernel issues HVCs from two functions. We denote them as main_sptm_gate and special_sptm_gate.

Main Secure Kernel SPTM Gate We identify this as the regular SPTM caller from the SK. x16 is parameterized comparable to the earlier subsystem calls from XNU. More precisely, it is set as follows:

Subsystem 2 5 different parameters, with the following x16 values: 0x200000000 – 0x200000004

Subsystem 4 2 different parameters, with the following x16 values: 0x400000000 – 0x400000001

Based on the fact that the SK seems to be the only caller using these subsystems, we assume these to be dedicated SK subsystems. This is in line with previous reverse engineering of the SK [42].

Special Secure Kernel SPTM Gate This special gate is surrounded by a function that zeroes general-purpose registers `x2` to `x15`, and `x17` to `x18`. It also completely zeroes the NEON vectors [14]. There are three invocations to this:

- Call with `x16 = 0xff00000000`
- Call with `x16 = 0xfe00000000`, in this case `x1` is also zeroed
- Call with `x16 = 0xfd00000000`, in this case `x1` is also zeroed

Without examining the SPTM handling side of these calls, we cannot yet deduce any clear meaning from them. However, based on the aforementioned calling preparation, we can assume this is not normal calling behavior, but probably handles unexpected errors or similar issues.

5.3.3 Calls from TXM

SPTM `x16` parameterization is also found in TXM, preceding SVC executions. This appears to be in line with TXM running in GL0, which we will examine more in-depth in Chapter 8. We find a single used SPTM gate in TXM, which we denote `core_SPTM_gate`. It is called with the following parameters:

- **Subsystem 1:** 4 different parameters, with the following `x16` values: `0x10000001` – `0x10000004`
- Call with `x16 = 0xff00000000`
- Call with `x16 = 0xfe00000000`
- Call with `x16 = 0xfd00000000`

Similar "special" calls outside of the standard subsystem numeration are found, as in the previous paragraph on SK SPTM gates. This supports the theory that these are special error-handling calls.

5.3.4 Calling SPTM – A Look into SPTM Headers

This preliminary analysis provided us with a fairly extensive view of interaction with SPTM. We can further this by incorporating header file releases by Apple, which were published with the `MacOSX15.5.sdk` release in April 2025 [3]. The Software Development Kit (SDK) contains three SPTM-specific header files, namely `debug_headers.h`, `sptm_common.h` and `sptm_xnu.h` located at `System/Library/Frameworks/Kernel.framework/Versions/A/platform/sptm`.

`sptm_common.h` provides us with key additional and "official" information on SPTM calling. The following provides insight into the parameterization and generally supports previous results. In Figure 5.3, we see a type description for the `sptm_dispatch_target_t` type, which we assume is the data type used in all SPTM calls and is stored in register `x16` for cross-component communication.

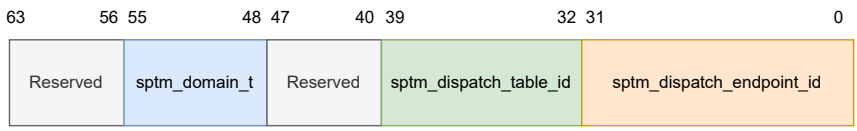


Figure 5.3: `sptm_dispatch_target_t`, adapted from `sptm_common.h`.

Regarding the `sptm_domain_t`, `sptm_dispatch_table_id` and `sptm_dispatch_endpoint_id` types used in Figure 5.3, we can glean significant information from the relevant header files.

sptm_domain_t Based on the `stpm_common.h` header, we find five SPTM domains, namely `SPTM_DOMAIN`, `XNU_DOMAIN`, `TXM_DOMAIN`, `SK_DOMAIN` and `XNU_HIB_DOMAIN` (see Section A.2). We assume `XNU_HIB_DOMAIN` to be the XNU hibernation domain[3]

Through the previous reverse engineering, we identified the relevant bits (48-55) as being set in only three specific calls from XNU. The special cases are invocations of `exclaves_enter`, `exclaves_bootinfo`, and `txm_enter` from XNU (see Section 5.3.1). For the first two, these specific domain bits were set to `0x3`, corresponds to the `SK_DOMAIN`. This aligns with our assumption that calls to Exclaves are managed by Secure Kernel as an intermediary between Exclaves and SPTM/XNU.

For the TXM call, the domain bit was set to `0x2`, which corresponds to the `TXM_DOMAIN`, and is in line with our assumption that this calls into TXM. All other calls detected so far are made with the domain entry set to `0x0`, indicating that the called domain is `SPTM_DOMAIN`, which we assume to invoke general SPTM services.

Although there is no public information available on the concept of SPTM domains in Apple's operating systems, we can make some assumptions based on the analysis of the header file. It appears that domains scope execution context and separate different parts of the operating system. They are not equivalent to privilege levels, but instead provide a new level of differentiation. We assume that SPTM domains are introduced to scope different security-relevant components from each other, thereby limiting access to each other's resources and reducing the risk of full system compromise. We will look into this scoping based on domains in Section 5.8.

sptm_dispatch_table_id_t What we have previously labeled as subsystems, as indicated by previous research, are in fact IDs of dispatch tables (see Section A.3). They are listed in `sptm_xnu.h`. This is in line with what we determined via reverse engineering on the SPTM dispatching process and will be further shown in Section 5.3.5.

This does, however, confirm our previous "subsystem" mapping, which we now know are dispatch table IDs. As determined, dispatch table ID 3 corresponds to `T8110dart`, table ID 5 to `SART`, table ID 6 to `NVMe` and table ID 7 to `UAT`. Interestingly, table ID 9 for the `CPUTRACE` system cannot be confirmed by this, as ID 9 is found to be `SPTM_DISPATCH_TABLE_RESERVED`. This might be an indication that this dispatch table is still under development or not supported by all devices.

We further assume that dispatch tables are assigned to specific called domains, with the listing seen in Section A.3 being `SPTM_DOMAIN` specific. This is indicated by the usage of these dispatch tables in calls with the domain parameter set to `SPTM_DOMAIN`. We will find this assumption validated in our analysis of dispatch table registration (see Section 5.3.5).

sptm_dispatch_endpoint_id_t The lower 32 bits of the parameter seem to encode the specific function/endpoint call. This aligns with what has been discovered so far. We will later use the now-available mapping for what we assume to be the `XNU_BOOTSTRAP` dispatch table (Section A.4) to infer SPTM usage and functionality. We can assume these endpoint IDs to be defined for specific dispatch table IDs in specific called domains.

[3] XNU hibernation is tasked with "suspending the entire system state to RAM", see `doc/lifecycle/hibernation.md` in XNU open-source code.

5.3.5 SPTM Dispatch Tables – Registration

SPTM is invoked in a variety of different ways. It offers a custom portfolio of available functions for different calling contexts. These different contexts are realized by domains and dispatch tables, both of which are part of the SPTM parameterization, as shown in Figure 5.3. SPTM supports a variety of different dispatch tables (Section A.3). The general dispatch structure is illustrated in Figure 5.4.

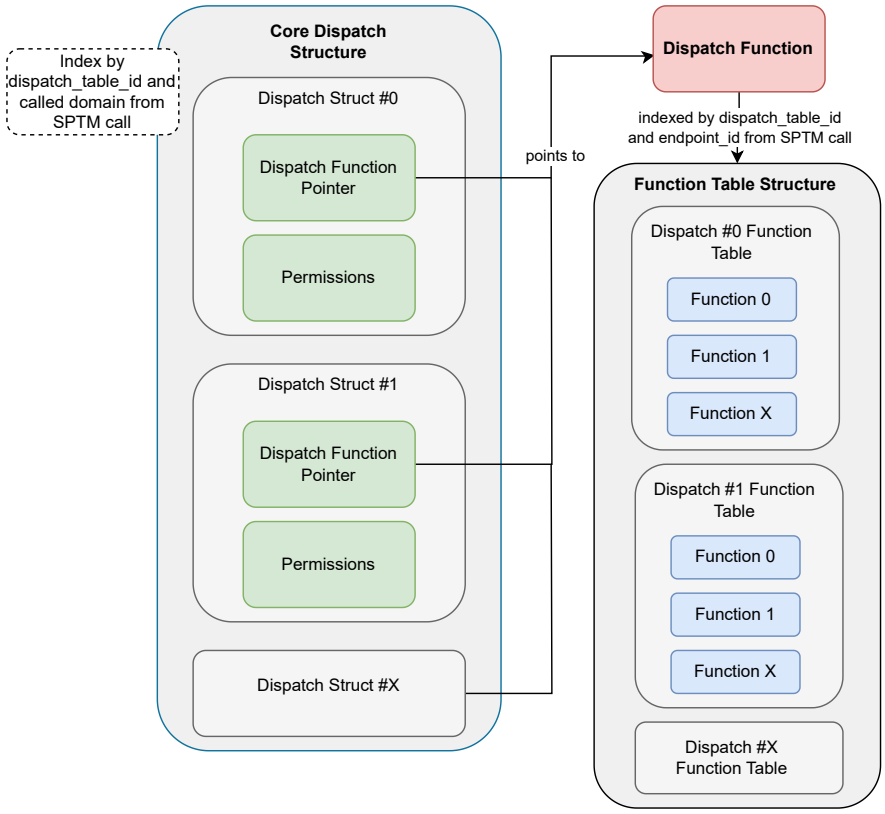


Figure 5.4: Function dispatching structure found in the SPTM binary. The *Core Dispatch Structure* holds multiple *Dispatch Structs*, and is indexed by the called domain and the dispatch table ID extracted from the SPTM call. Each entry contains a *Dispatch Function Pointer* and *Permissions*, with the permissions being used to validate the call to it, and the dispatch function pointer containing a pointer to a function handling the actual function dispatch. The *Dispatch Function* accesses the corresponding *Dispatch Function Table* based on the SPTM call endpoint value, and indexes the specific function based on the SPTM call dispatch table.

Significant parts of the setup and registration of the dispatch logic are performed at runtime and are therefore not fully recoverable from a static analysis of the SPTM binary; however, key parts for request handling are still detectable. We find a structure we deem the *Function Table Structure*, which holds four pointers to dispatch function tables in the binary. These four table pointers are the `XNU_BOOTSTRAP`, `TXM_BOOTSTRAP`, `SK_BOOTSTRAP` and `HIB` table respectively. These names were inferred by index matching the table entries with the SPTM headers (see Section A.3), and from the calling context.

With regards to the *Core Dispatch Structure*, we presume a pointer to it to reside at a specific point in memory, which we call *Core Dispatch Structure Pointer*. It is, however, not

initialized in the binary, and we presume it gets set up at runtime. Its location is clear via its usage in `sptm_register_dispatch_table`. This function name is known from the debug string usage within its body. We further know the signature of this function, as it is included in `sptm_common.h`. The signature is as follows:

```
void sptm_register_dispatch_table(
  sptm_dispatch_table_id_t table_id,
  sptm_vaddr_t dispatch_entry,
  uint64_t permissions);
```

From this signature and the function body, we can verify that the `dispatch_entry`, which we have denoted `dispatch_function_pointer`, and the specific access permissions are indeed written to the specific *Dispatch Struct*. During population, the *Core Dispatch Structure* is indexed and populated not only based on the dispatch table ID, but also on what we presume to be the caller domain, stored in the thread-identifying information. This makes sense as calling into different domains should result in different dispatch tables.

Registering for XNU, TXM, SK, and HIB For these four dispatch "systems", the *Dispatch Function Table* pointer is statically set up in the *Function Table Structure*. Therefore, they are directly set up with regards to their permission and *Dispatch Function* via calls to `sptm_register_dispatch_tables` from a function we presume to be `init_xnu_ro_data` based on debug string usage. All except the `HIB` table are registered unconditionally, whilst for the `HIB` table, a flag in memory is checked. This might be based on system support for hibernation, which is not provided for all Apple devices supporting SPTM (MacBooks vs. iPhones). From the invocations, we can infer the permissions set for each table.

```
1 sptm_register_dispatch_table(0, sptm_dispatch, 2);
2 sptm_register_dispatch_table(1, sptm_dispatch, 4);
3 sptm_register_dispatch_table(2, sptm_dispatch, 8);
4 if (hib_flag == '\x01') {
5     sptm_register_dispatch_table(10, sptm_dispatch, 0x10);
6 }
```

Listing 5.4: SPTM dispatch table registration from `init_xnu_ro_data`. The source code was disassembled by Ghidra.

We can extract the specific bit set in the permission flag for each call and compare it to the corresponding domain ID (see Table A.2). Additionally, we have further mapped the dispatch table ID to the corresponding table name (see Table A.3).

Table ID	Matching Table Name	Permission	Permission Bit Set	Matching Domain
0	XNU_BOOTSTRAP	0x2	1	XNU_DOMAIN
1	TXM_BOOTSTRAP	0x4	2	TXM_DOMAIN
2	SK_BOOTSTRAP	0x8	3	SK_DOMAIN
10	HIB	0x10	4	XNU_HIB_DOMAIN

Table 5.2: SPTM dispatch table registrations and permissions from `init_xnu_ro_data`.

We will later examine the exact usage of these permissions during SPTM call invocations, but it appears logical that only corresponding domains should have access to their specific dispatching logic.

Registering for Input/Output Memory Management Units (IOMMUs) SPTM not only serves as the access permissions handler for XNU, TXM, and SK, but also for various different IOMMUs. Therefore, there is a dedicated function for bootstrapping IOMMUs in SPTM, which we have labeled `IOMMU_bootstrap`, inferred by panic string usage. It is called from a large Input/Output Memory Management Unit (IOMMU) setup function. This calling function invoked `IOMMU_bootstrap` with five different integer parameters. We can map these integers to the corresponding IOMMU ID (see Table A.5). This confirms that the bootstrap function is called for SART, NVME, UAT, and DART_T8020. It is further called for IOMMU ID 7, which we do not find in the respective header file. This might indicate that support for the IOMMU has been removed from SPTM.

At a high level, the bootstrap function validates the IOMMU and reserves memory frames for it. The reserved memory frames are typed `SPTM_IOMMU_BOOTSTRAP`. The number of frames to reserve seems to be dependent on an IOMMU-specific global structure, which SPTM accesses based on the IOMMU ID.

The actual dispatch table registration for IOMMUs is performed via a call to `register_iommu`. This call is parameterized with the IOMMU ID, the corresponding dispatch table ID and dispatch table, and permissions that are set based on the IOMMU ID. Permissions are set to 0x2 for all but the NVME registration. In that case, the permissions are set to 0x12. A listing of the permissions set can be seen in Table 5.3.

IOMMU ID	IOMMU name	Permission	Permission Bits Set	Matching Domain
1	SART	0x2	1	XNU_DOMAIN
2	NVME	0x12	2,3	TXM_DOMAIN, SK_DOMAIN
3	UAT	0x2	1	XNU_DOMAIN
5	DART_T8110	0x2	1	XNU_DOMAIN
7	?	0x2	1	XNU_DOMAIN

Table 5.3: IOMMU ID and corresponding permissions set in the dispatch table registration.

It appears that XNU is allowed to call into the dispatching tables for all but the NVME IOMMU, which seems to be only valid for TXM and SK. The implications of this are still unclear.

This `register_iommu` function invokes the previously looked at `sptm_register_dispatch_table` function to place the relevant dispatching information into the `Core Dispatch Structure`, and additionally sets the `Function Table Structure` for the specific IOMMU. A disassembled and labeled version of the `IOMMU_bootstrap` function can be found in Section A.6.

5.4 SPTM Request Handling

As SPTM can be invoked from a variety of different contexts, it must employ multiple means of handling incoming requests. Regardless of the request's origin, SPTM employs an extensive internal dispatching regime, with which it tracks the execution state and allows future request invocations. We will examine the internal dispatching in Section 5.4.1

before analyzing the actual handling of calls invoked from both Exception Level (EL) and GL components.

5.4.1 Internal Dispatching

SPTM clients can call into it in different ways: `GENTER`, SVC, and HVC. Most of these calls are, in one way or another, forwarded to a core request handling functionality within SPTM. Before examining the handling of special invocations from other components, we will dissect this core function, which we denote `CORE_SPTM_FUNCTION`.

Based on context, the function is parameterized with an `event_type`, and a 64-bit call parameter, which we assume to be of type `sptm_dispatch_target_t` (see Figure 5.3). The function does some initial verification on the `event_id`[4] and the state stored in the current thread identifying information (TPIDR register)[5]. If these checks are passed, an entry from a table is retrieved from an in-memory transition table. This retrieval is based on the `event_id` and current state. As a first layer of implemented security, the `next_state` field is checked from the transition table entry and verified to be within bounds. In a similar fashion, a so-called `state_transition_action` (name inferred from panic string usage) is retrieved and verified to be set in the table entry. Any failure on these conditions indicates a forbidden transition attempt, leading to an immediate panic. Furthermore, a flag is retrieved from the entry, and a fourth field denoting an SPTM domain is written to the TPIDR caller domain field. An assumed transition table entry struct based on the fields accessed so far is shown in Figure 5.5.

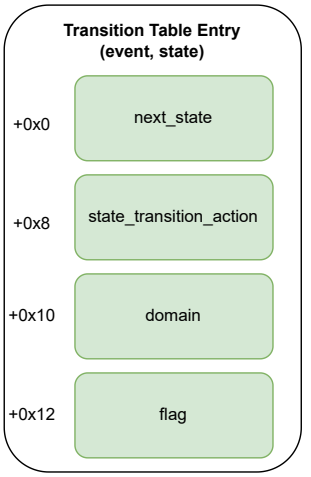


Figure 5.5: A transition table entry as retrieved by the `SPTM_CORE_FUNCTION`, indexed by the `event` parameter and `state` retrieved from thread identifying information.

An overview of the values retrieved from the transition structure is presented in Section A.7. If the low bit is not set in the aforementioned flag, the `dispatch_entry_point` is set to be empty. This might correspond to transitions that require no further dispatching. Otherwise, it is calculated based on the `sptm_dispatch_target_t` parameter. The calculation is shown below.

[4]Provides us with the knowledge that the `event_id` must be at most 14.
[5]Provides us with the knowledge that the state must be at most 22.

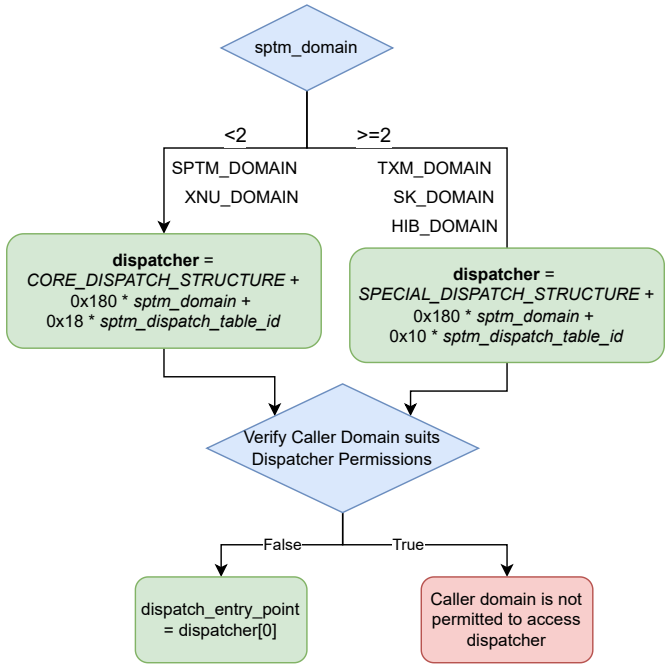


Figure 5.6: Calculation of the `dispatch_entry_point in` in `SPTM_CORE_FUNCTION`. Depending on the `sptm_domain_t` value from the input `sptm_dispatch_target_t` value, a dispatcher is retrieved from memory.

Based on the domain specified in the parameter, a different dispatcher is retrieved. If the domain is `XNU_DOMAIN` or `SPTM_DOMAIN`, we retrieve it from the `CORE_DISPATCHER_STRUCTURE`, for which we dissected the population in Section 5.3.5. Otherwise, we retrieve it from a specific memory address we have denoted `SPECIAL_DISPATCH_STRUCTURE`. Our analysis of TXM and SK will show that they register their respective dispatch tables at that location.

Seeing this differentiation, we can look for specific SPTM calls in Section 5.3.1 as a short interjection. Interestingly, we observe that for nearly all calls, the `sptm_domain` field is not set, hence corresponding to the `SPTM_DOMAIN` value. We find two exceptions in the `txm_enter` SPTM gate and the `sk_enter` SPTM gate, which are performed with the respective domain set in the request. Logically, we can assume that for these calls, SPTM acts only as an intermediary, and "forwards" them to the actual target.

As can be seen in Figure 5.6, the function further validates that the permissions set for the dispatcher retrieved are in line with the caller domain that has been set based on the domain flag in the transition structure. This ensures that only intended caller domains can perform specific state transactions.

After a variety of further conditional checks, the actual state transition action is performed, parameterized with the `sptm_dispatch_target_t` parameter, and the dispatch entry point calculated. With this, the actual request handling is performed, which will be looked at in Section 5.4.2 and Section 5.4.3.

Internal Dispatch State Machine For state/event combinations, we find some, but not nearly all of the transition actions implemented. From this, we can deduce that there is a relatively rigid internal state dispatching logic that has to be adhered to.

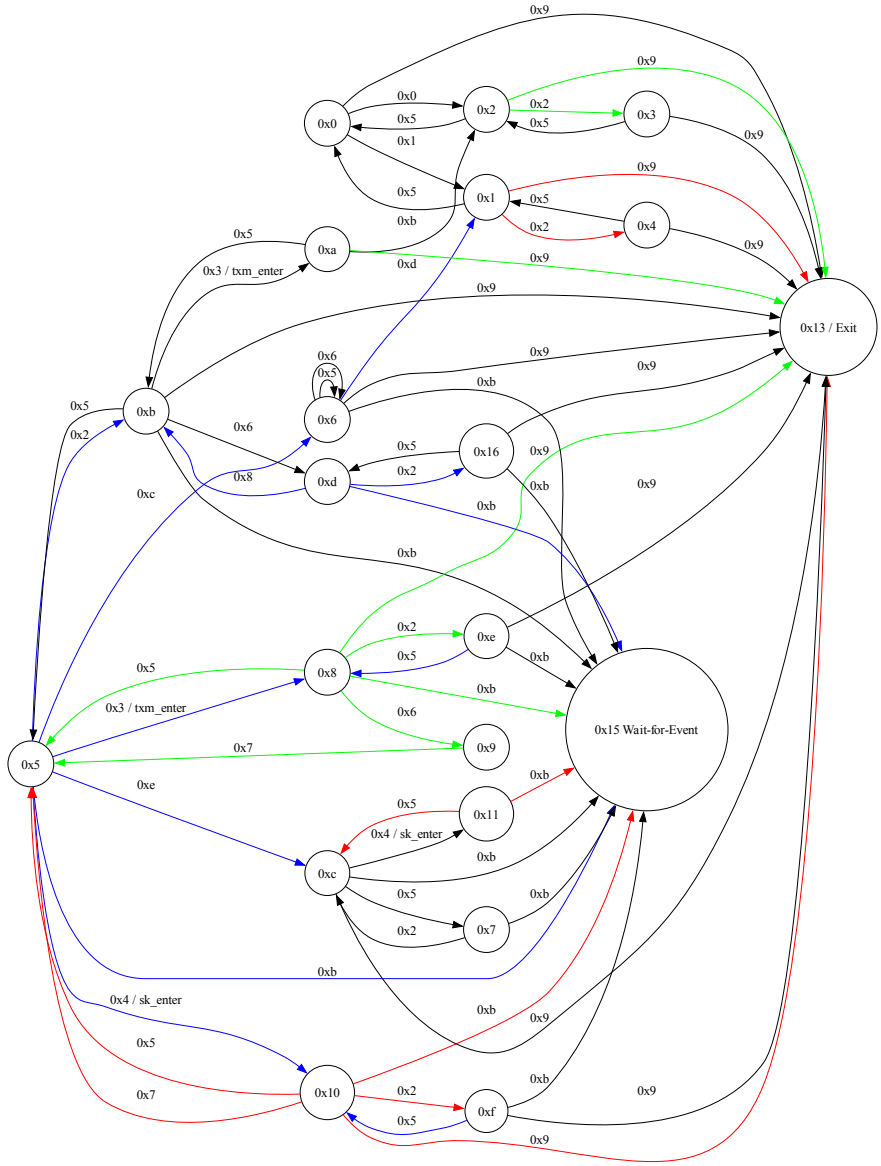


Figure 5.7: SPTM-internal state graph. Vertices show states, and edges show events. The graph is colored based on the domain flag set in the transition structure entry and is shown in Section A.7. SK_DOMAIN is colored red, SPTM_DOMAIN is colored black, TXM_DOMAIN is colored green, and XNU_DOMAIN is colored blue. States and events have partially been labeled based on our reverse engineering.

The complexity of the state transition actions implemented is highly diverse. Some are fairly simple, involving only system register writes, while others require dispatching and appear to trigger additional state transitions after implementing functionality. An enumeration of valid transitions can be found in Section A.7. The table lists the transition action label, the next state entry, the domain entry (which gets written to the TPIDR

caller domain field), the flag entry, and the corresponding domain name for the respective domain ID for valid state event combinations. Valid combinations are those that have a transition action set.

Figure 5.7 presents a visual representation of valid transitions from Section A.7. Vertices are state IDs, and edges are event IDs. We can partially infer the internal SPTM states from this graph. It is directly observable that both state `0x13` and `0x15` act as sinks in this state machine, being transitioned into, but with no valid outgoing transitions.

Towards state `0x13` we find transitions through event `0x9` from the following states: `0x0`, `0x1`, `0x2`, `0x3`, `0x4`, `0x6`, `0x8`, `0xa`, `0xb`, `0xc`, `0xe`, `0xf`, `0x10`, `0x16`. For states `0x0`, `0x2`, `0x3`, `0x4` we find a call to a low-level logger function `low_level_logger`. This function appears to write a string to a hardware buffer. The string written is *[PANIC DURING BOOTSTRAP]*. The same happens for state `0x1`, but for the string *[SK BOOTSTRAP PANIC]*. After these writes, both transitions switch into a *Wait-for-Event*-loop. From all other states, event `0x9` leads towards a `GEXIT` instruction, with system state manipulation performed before. To summarize, state `0x13` appears to be an internal state transition that occurs when exiting SPTM. These exits are performed via event ID `0x9`. Exits can either be a direct result of a panic or due to other unknown reasons.

With regards to state `0x15`, we find transitions towards it from the following states: `0x5`, `0x6`, `0x7`, `0x8`, `0xb`, `0xc`, `0xd`, `0xe`, `0xf`, `0x10`, `0x11`, `0x16`. All these transitions invoke the same function, which zeroes a currently unknown system register with encoding (`0x0`, `0x3`, `0x3`, `0xb`, `0x4`), and then enters a *Wait-for-Event*-loop. This might indicate that state 0x13 is a "successfully finished" state for SPTM internal tasks that do not require returning to a caller.

We have further labeled edges we determined to be exclusively used for entering SK and TXM accordingly. Apart from that, we have not performed a deeper analysis of this internal state machine, and leave this as future work.

5.4.2 GENTER

We have previously determined that the SPTM `GENTER` handler is set via a register write to the `GXF_ENTRY_EL1` register, which points to the handler. The address points to a function we denoted `GXF_entry_point`. The function performs large context-setting operations and conditionally branches based on the `ESR_GL1` [6]. The register stores information on exceptions taken to GL1. The exact working of this handling is unclear at present, as we do not expect the function to be called in a normal exception handling path, but rather only on GXF entry via `GENTER`. The key function call here is, however, a branch to `genter_dispatch_entry`, parameterized with the received `x16` `sptm_dispatch_target_t` value. The full function can be found in Section A.8.

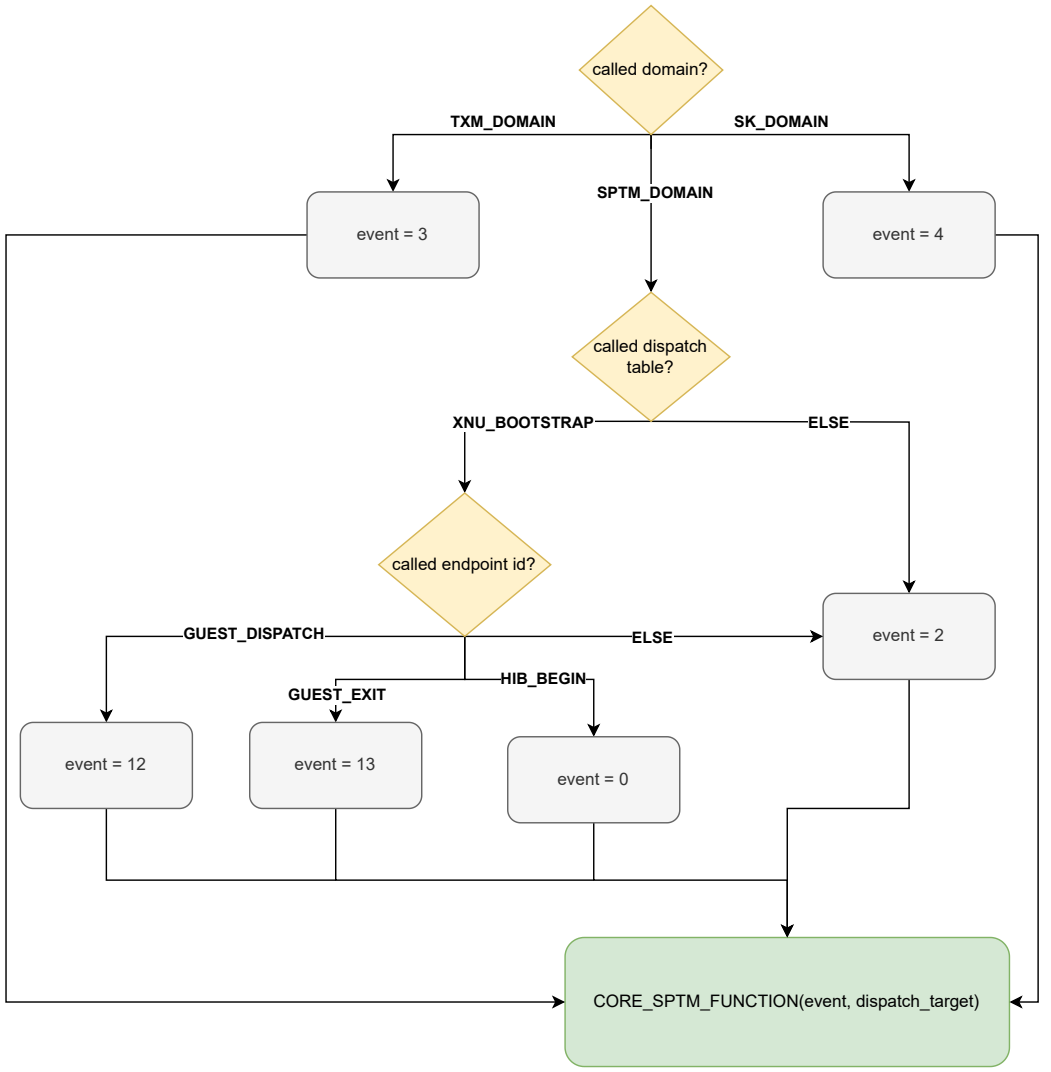


Figure 5.8: Control flow of `genter_dispatch_entry` for retrieving the internal event code based on `GENTER` input parameter `dispatch_target`.

Figure 5.8 shows a schematic of the `genter_dispatch_entry` control flow. We find event 3 to be the internal event for calling into TXM, and event 4 to be the internal event for calling into the Secure Kernel, respectively. For calls into the SPTM domain, event 2 appears to be the default handling event, with special event codes being denoted to specific endpoint calls in the `XNU_BOOTSTRAP_TABLE`. The actual dispatch handling is performed via a call to `CORE_SPTM_FUNCTION`, which we described in Section 5.4.1.

5.4.3 SVC/HVC Handling

We have previously looked at the request handling for SPTM requests from the EL client XNU. These were found to be performed via `GENTER` to switch context into GXF. For SPTM clients already running in GXF, these calls are performed via SVC or HVC instructions.

5.4.3.1 GL0 SVC Rerouting to SPTM

We have found GL clients of SPTM to call into it with SVC or HVC instructions, depending on their respective guarded level. Whilst it is standard for Secure Kernel to call into SPTM via HVCs from GL1, SVC instructions executed at GL0 would usually be routed to GL1, rather than to SPTM in GL2. We find that such SVCs are conditionally trapped towards SPTM in GL2 by the usage of the Hypervisor Configuration Register EL2 (HCR_EL2). The register provides configuration for virtualization, and appears to also affect SPTM in GL2. The relevant register field is the Trap General Exceptions (TGEs) bit at position 27. If set, in standard Arm exceptions routed to EL1 are rerouted to EL2 [11]. We can assume that for components running in GLs, this performs exception rerouting to GL2.

We have not reverse-engineered the conditional setting of this flag as of now. The exact inner working of the request handling logic is still unknown, considering we will show in Chapter 6 that GL0 components actually directly call into Secure Kernel in GL1. Based on the clear SPTM parameterization of SPTM calls, we assume these calls are routed to SPTM. The exact handling mechanisms for allowing GL0 components to call into Secure Kernel at GL1 via SVCs and TXM calling into SPTM at GL2 via SVCs at the same time have yet to be discovered.

5.4.3.2 Request Handling

GL clients of SPTM are found to call into it via SVC and HVC instructions. Based on the initial analysis by Dataflow Forensics [25] and the aforementioned GXF setup functions, it is possible to deduce information on the SVC/HVC handling by SPTM.

The function `gxf_setup_late` includes a system register write to `VBAR_GL1`. As reasonably assumed by Dataflow Forensics, this appears to work in a similar fashion to the Arm standard register `VBAR_EL1`, and therefore contains the base address for various exception handlers [18]. More specific information on this can be found in the Arm documentation on exception handling [16]. We expect a synchronous exception handler for exceptions taken from lower levels at `VBAR_GL1 + 0x400`. At the specified address, we find the branch to the actual handler we denoted `synchronous_exception_handler_from_lower`.

The handler implements complex functionality regarding exception call handling. Request handling is differentiated based on the aforementioned hypervisor configuration register. The full listing of the handler can be found in Section A.9. We find the handler to implement functionality for both SVC and HVC handling.

5.4.3.3 SVC Exception Handling

A conceptual overview of SPTM SVC handling can be seen in Figure 5.9. Furthermore, we show an excerpt on SVC handling in `synchronous_exception_from_lower` in Section A.10.

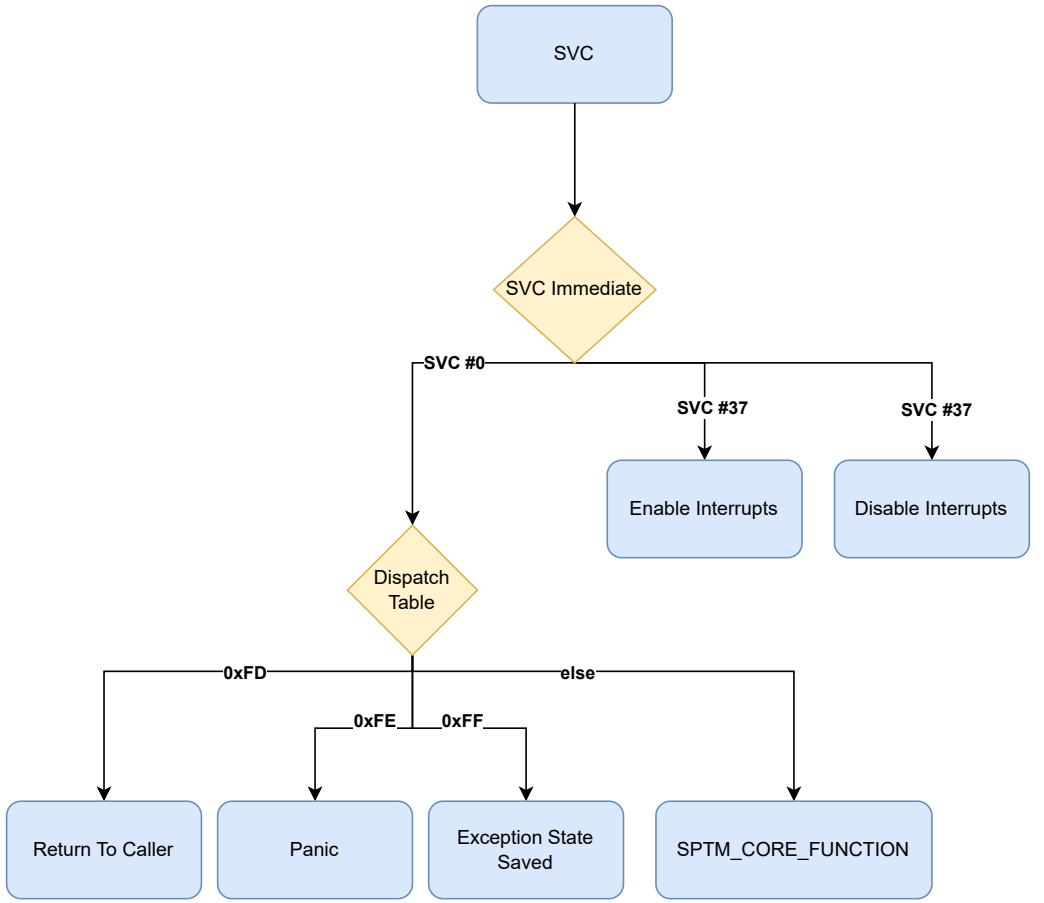


Figure 5.9: Conceptual SVC exception handling control flow of SPTM.

SVC #37 & SVC #38 – Interrupts The workings of these two SVCs have already been thoroughly reverse-engineered by Dataflow Forensics [25], and our analysis of a newer firmware version verifies their results; therefore, a comparable approach is employed in the current SPTM. Listing 5.5 shows the handler for an SVC #37 call.

```
                    SVC_37_HANDLER
fffffff027081fb4 68 fa 3e d5     mrs        x8,SPSR_GL1
fffffff027081fb8 09 38 80 d2     mov        x9,#0x1c0
fffffff027081fbc 08 01 29 8a     bic        x8,x8,x9
fffffff027081fc0 68 fa 1e d5     msr        SPSR_GL1,x8
fffffff027081fc4 e0 03 9f d6     eret
fffffff027081fc8 a0 d5 9b d2     mov        x0,#0xdead
```

Listing 5.5: SVC #37 handler in SPTM

The system register operated on is the Saved Program Status Register (SPSR) (GL1). The operation performed on it sets the A, I, and F bits of the register to zero. These bits, if set,

mask SError exceptions, IRQ exceptions, and FIQ exceptions [7]. We therefore confirm that SVC #37 enables interrupts. We assume this to be called when exiting critical code execution, which is usually protected from disruptions by disabling interrupts. Analogue to this, SVC # 38 disables all interrupts by setting the A, I, and F bits of the SPSR (GL1). The respective handler can be seen in Listing 5.6.

```
                    SVC_38_HANDLER
fffffff027081fd4 68 fa 3e d5     mrs        x8,SPSR_GL1
fffffff027081fd8 08 09 7a b2     orr        x8,x8,#0x1c0
fffffff027081fdc 68 fa 1e d5     msr        SPSR_GL1,x8
fffffff027081fe0 e0 03 9f d6     eret
fffffff027081fe4 a0 d5 9b d2     mov        x0,#0xdead
```

Listing 5.6: SVC #38 handler in SPTM

Both of these handlers end the exception handling via the `eret` instruction, returning to the caller of the SVC. Therefore, we return to a lower-privileged process here. We find such calls solely performed by TXM.

SVC #0 – Dispatch Targets SVC #0 is used as the main call to invoke SPTM functionality from its GL0 clients. The handling differentiates based on the provided dispatch table ID in the `x16 sptm_dispatch_target_t` parameter. We know from Table A.3 that `0xFF`, `0xFE` and `0xFD` denote special dispatch table IDs that invoke control functions. For any other dispatch table ID, we find the call forwarded to `CORE_SPTM_HANDLER` after performing context preparation.

HVC Exception Handling

Comparable to SVC exception handling, HVC #0 calls are handled depending on the set dispatch table ID. The same control functions as for SVC calls can be invoked via dispatch table ID values `0xFF`, `0xFE`, and `0xFD`. For any other value, the handler dispatches to `CORE_SPTM_FUNCTION` via another function call. There is no check for other immediate HVC values, so we assume GL1 clients of SPTM, namely Secure Kernel, do not either rely on interrupt disabling via SPTM or direct this towards lower-level components.

5.5 SPTM Functions

Whilst SPTM performs a variety of different tasks for which it can be called into by its clients, frame retyping and page mapping appear to be the most central ones. Frame retyping enables SPTM clients to modify the underlying SPTM frame type of memory frames according to a predefined rule set. Regarding page mapping, SPTM serves as the sole authority and component privileged to modify page table data. Memory mappings can be requested by its clients, with SPTM performing them based on a rule set built upon the aforementioned SPTM frame types. We will examine these functionalities in more detail in the following sections.

5.6 SPTM Frame Retyping – Overview

SPTM introduces so-called frame types. A conclusive list of all available SPTM frame types can be seen in Section A.11. Types are separated into SPTM-owned, XNU-owned, TXM-owned, and SK-owned memory types. As we will demonstrate in Section 5.8, memory frame types are a crucial building block for memory mapping and system security. SPTM performs frame retyping via the `retype` function, which we find registered in the bootstrap table for XNU, TXM, and Secure Kernel, respectively. They all, therefore, appear to request frame retyping.

We find SPTM frame types to realize memory frame ownership, with owners of a page being one of the known SPTM domains. Ownership of memory frames via the SPTM type is required to map memory into them via `sptm_map_page`, which is the sole function usable for page table alterations.

5.6.1 SPTM Frame Retyping – Calling from XNU

We find XNU invocations to SPTM via `sptm_retype`, and can infer on the function signature based on the XNU open-source code. The signature as extracted from XNU can be seen below:

```
void sptm_retype(pmap_paddr_t physical_address, sptm_frame_type_t
    previous_type, sptm_frame_type_t new_type, sptm_retype_params_t
    retype_params)
```

The function is parameterized with the physical address of the frame to retype, the current type, the new type, and retyping parameters. The fact that the current page type is provided suggests a list of allowed frame retyping transitions, which we will confirm through our analysis. As of now, we are uncertain why the current frame type must be provided by the caller. As SPTM is tracking frame types, it is aware of the specified frame's type.

5.6.2 SPTM Frame Retyping – Fundamentals

We find SPTM to perform retyping on memory frames. Recalling Section 2.1.2, a memory frame refers to a chunk of physical address space. It appears that SPTM is tasked with tracking physical memory in a frame table. A frame table typically refers to a table containing a list of all physical memory frames, along with additional state information on them [52]. Entries in SPTM's frame table are referred to as Frame Table Entrys (FTEs). We find the underlying frame type of a frame stored in the relevant FTE.

5.7 SPTM Frame Retyping – In-Depth

Retyping of memory frames is the key task of SPTM. On the SPTM side, it is performed via the `retype` function. We know it to be invoked from XNU with `x16` parameter value 0x1 by the SPTM function ID mapping (see Section A.4). As already determined, `sptm_retype` is called on a physical address, with a current type, a new type, and "sptm retype_params" (e.g. `osfmk/sptm/pmap.c`, line 6328-6341). This is also confirmed in the `sptm_common.h` header. We further know `retype_params` to store "Type-specific information set at retype time" (see `sptm_common.h` header). Based on invocations observed in the XNU open-source code, these parameters are frequently left empty, indicating that they are not required for most type transitions.

5.7.1 Handling Retyping Requests

While we can infer on the inner workings of SPTM `retype`, it is a highly complex function for which we have not fully reverse-engineered the entire control flow. Instead of a complete understanding of the underlying function, we are more interested in implemented security mechanisms, and we will put our focus on those.

5.7.1.1 Early Validation

On call, the function performs early validation, confirming the relevant preconditions for executing the function. It initially validates that the provided physical address is actually a managed frame. This is done by checking if it is within the globally stored relevant bounds. We can infer these bounds from error handling within another function[6]. The next conditional check verifies that the new type is at most 62, which is in line with the SPTM types found in the SPTM headers (see Section A.11). What follows is what we assume to be a FTE retrieval, with the FTE holding various flags and information on the frame it describes. An "in use" flag of this descriptor is checked and set, whilst panicking if already set[7]. The early input validation is illustrated in Figure 5.10.

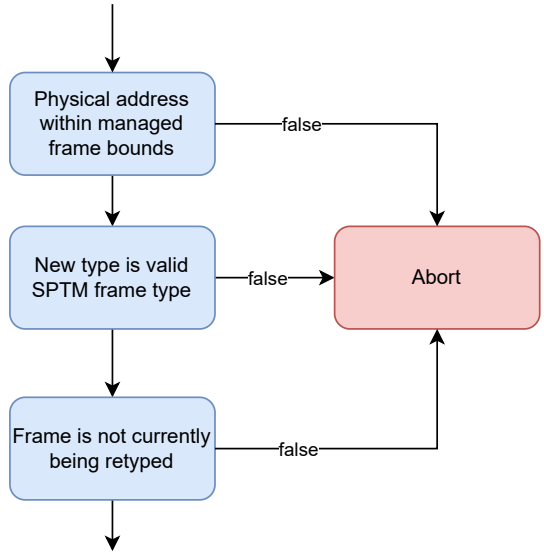


Figure 5.10: Early validation in `retype`.

5.7.1.2 Type-Specific Call Validity Checks

After initial verification passes, more type-specific call validity checks are performed. The `retype` function validates the caller domain with respect to the frame's current type. This check is only executed if the current frame type is not `SPTM_UNTYPED`, which indicates that retypes to such frames are not limited based on the caller domain. For this check, a single byte is indexed from a global structure we denoted `AllowedCallerDomains`,

[6] Error handling in `FUN_ffffff0270b4118`.

[7] The meaning of this flag was inferable from later usage and debug strings, especially the "Attempted to update PAPT permissions outside of a retype() operation, this is unsafe." string after flag usage at `0xffffff0270b2aa8`.

based on the frame type. A full listing of the allowed caller domains for specific frame types can be found in Section A.12. This is largely in line with expectations, with each domain being able to retype memory frames typed to one of its respective SPTM types. The exception to this seems to be the `XNU_TAG_STORAGE` type, for which the allowed domain is the SPTM domain.

A further key check here is comparing the function parameter `previous_type` with the actual type in the FTE. If they do not match, the retype fails. This mitigation prevents SPTM retyping of frames in unexpected conditions, and also protects against tampering with the previous conditional check by faking the original frame type. The caller validity check is illustrated in Figure 5.11.

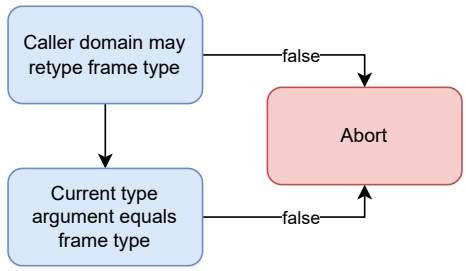


Figure 5.11: Caller validity check in `retype`.

5.7.1.3 Retyping Validity Checks

After early validation and caller validity checks have been passed successfully, the first actual retyping validity check is performed. To achieve this, an entry is retrieved from a global table, which we denote as `SPTM_Retype_Global_Table`, indexed based on the current type of the frame. The retrieved value is shifted by the new type argument, and it is checked if, after this shift, the low bit is set, indicating an allowed retype operation. We find this to be classic validity checking logic, where a bit set at a specific position in the table entry value indicates a valid retype to that type. The full list of allowed frame retype operations can be seen in Table A.11. The conditional check can be seen in Listing 5.7.

```
1 if ((*(ulong *)(&SPTM_Retype_Global_Table + CurrentFrameTypeOffset) >>
   ↪  (new_type & 0x3f) & 1) == 0
2  ) {
3     DEBUG();
4 }
```

Listing 5.7: Retyping validity check based on the current frame type, retrieving and checking all allowed types to retype to. The source code was disassembled by Ghidra.

As expected, `SPTM_UNTYPED` is allowed to be retyped to any frame type. As we have previously seen in Section 5.7.1.2, for frames of type `SPTM_UNTYPED`, the caller domain is not considered, so we assume this to be the default frame type for which components running in other SPTM domains can retype memory to their domain-specific types.

After this transition validity check, a type-specific `type_out` function is retrieved from a structure we denote `RETYPE_Typeout_Structure`. The structure is indexed based on the current frame type. We find the existence of these `type_out` functions

supported by strings in the binary, where we see references to these functions for specific types.[8]Notably, for most entries this function is not defined. For frame types with typeout functions implemented, they are listed in Table A.12. The `type_out` functions generally implement further type-specific validity checking logic, which we find partially duplicates the retype validity checking performed in the retype function. Following this, a further entry is retrieved from a structure we denote `RETYPE_Flag_Structure`, indexed based on the current type. These type-specific flags are found in Section A.12. The FTE is altered based on the flag. The exact workings of this are still unclear, but we assume it will alter frame parameters if necessary for the new frame type.

Comparable to the `retype_out` operation performed before, a `retype_in` operation is then performed based on the new type. If defined, the function is parameterized with the FTE, the new type, the retyping parameters, and an output pointer. The listing of defined functions can also be found in Table A.12. The control flow to this point can be seen Figure 5.12.

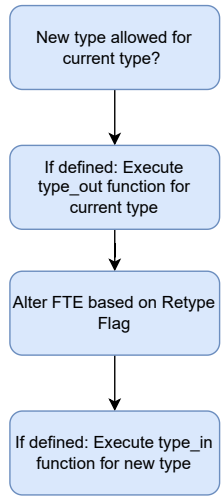


Figure 5.12: Control flow in SPTM `retype` after initial and caller validation passed.

5.7.1.4 PAPT SPRR Updating

Finally, we find a call to a method we denoted `PAPT_permission_update`. The name is inferred from assumed functionality and debug strings. Our understanding of this function call is limited. Based on our speculation, we assume it maps physical addresses back into virtual memory using the Physical Aperture Table (PAPT) and updates its SPRR index based on input parameters. The SPRR index to update is retrieved from the same global retyping structure we accessed before. We have denoted its base address as `SPRR_Index_From_Type`. The full listing of SPTM types to SPRR index mappings can be found in Section A.13.

[8]An example for this is the `sk_types_retype_out` string. We shall denote the corresponding function as `sk_types_retype_out`. Looking at references to the function, we find them at `RETYPE_Typeout_Structure` offset by multiples of types 59/60/61, respectively. Verifying with Section A.11, we find these values to in fact correspond to secure kernel types. Interestingly, the `SK_IO` type does not have this function called as retype out.

5.7.1.5 Finalizing the Call

The `retype` call is finalized by accessing and updating an in-memory SPTM data structure, which we assume to be responsible for tracking and enforcing type and permission metadata for memory frames. As SPTM appears to be the sole authority regarding memory frame management, it is logical to track the changes performed. There is currently no information available regarding these internal structures, and we have not yet reverse-engineered them.

5.7.2 SPTM Frame Retyping – Security Implications

Frame retyping via SPTM employs a strict ruleset and a variety of conditional checks to ensure that only a specific, pre-allowed set of retypings can occur. Generally speaking, frame retyping is scoped to occur within the same SPTM domain, with a few exceptions. The underlying security implications become clear when examining the second core SPTM functionality, specifically page mapping.

5.8 SPTM Page Mapping – Overview

We have previously discovered that one of SPTM's capabilities is the mapping of memory pages. This was indicated by the existence of an endpoint ID `MAP_PAGE` for the XNU bootstrap table (see Table 7.7). We find multiple calls to `sptm_map_page` in `osfmk/arm64/sptm/pmap.c` in the XNU open-source code, which helps us infer information about the calling parameters. The function signature is shown below:

```
sptm_return_t sptm_map_page(pmap_paddr_t ttep, vm_address_t va,
    pt_entry_t new_pte)
```

The function is usually called with the `ttep` parameter set to the `ttep` field of a Physical Map (PMAP). A physical map is a per-process structure holding information on address translation. The `ttep` field of a pmap is the "physical page of the root translation table" (see `osfmk/arm64/stpm/pmap.h`), that is, the physical address of the initial translation table to consult when trying to translate a virtual memory address to a physical address. The second function parameter is a virtual address, and the final parameter of type `pt_entry_t` is a PTE.

5.8.1 Core Call to `sptm_map_page`

Whilst `sptm_map_page` is invoked at a variety of locations, we find the core one in the XNU open-source code from `pmap_enter_pte` (`osfmk/arm64/sptm/pmap/pmap.c`). To provide context for the SPTM invocations, we trace back its callers to `pmap_enter_options` (`osfmk/arm64/sptm/pmap/pmap.c`). We know this function to be responsible for inserting translation table entries and introducing new virtual-to-physical address mappings [22]. The function calls an internal function that performs the actual page table entry creation, and then invokes `pmap_enter_pte` to update the created page table entry. As XNU is not entitled to write to page tables (recall Section 2.3.4), it needs to perform this update via the `sptm_map_page` invocation.

Recalling its function parameters, the `ttep` parameter in this case is the physical frame of the root translation table for the `pmap` into which XNU wishes to insert a page table entry. The `va` parameter holds the virtual address mapped by the newly created pointer, and the `new_pte` parameter stores the value to store in the new page table entry. This

`new_pte` is a block entry, and therefore directly holds the output physical address to map the virtual address to. The logic around this function is shown in Figure 5.13.

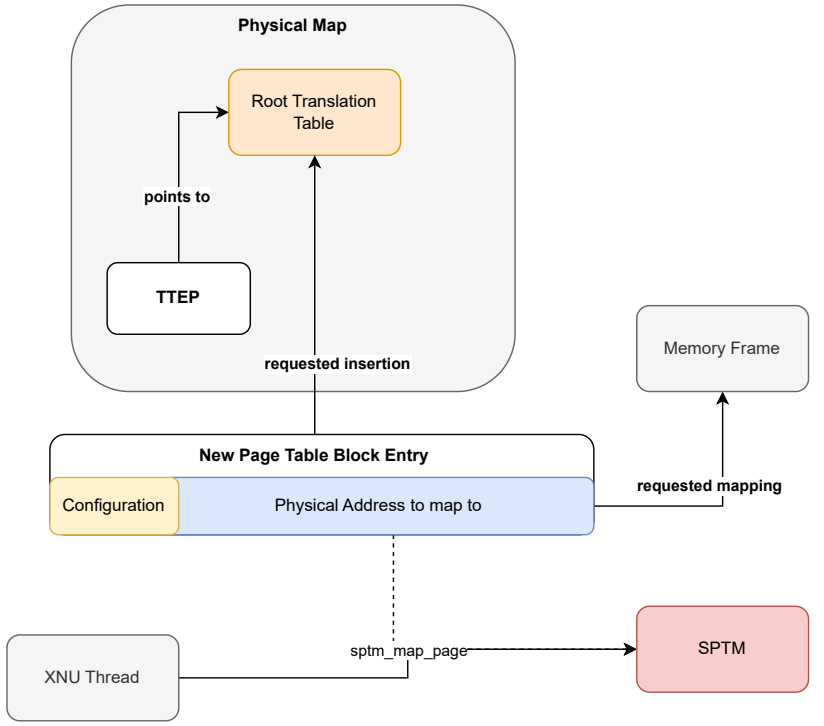


Figure 5.13: Schematic of the process leading to an invocation of `sptm_map_page`. A XNU thread attempts to create a new page table entry and insert it into a physical map with a mapping to a targeted physical frame.

5.8.2 `sptm_map_page` Security Mechanisms

We know `sptm_map_page` to perform the actual insertion of the page table entry into the requested memory mapping. It is handled on the SPTM side via `map_page`. This function is highly complex and performs a variety of conditional checks. Inferring its exact inner workings is made difficult by the lack of symbols and the use of strings in the code. We do not aim to provide a comprehensive explanation of its inner workings and make no claim to completeness regarding security features, but instead attempt to highlight key mechanisms. We know from the previously examined SPTM headers that `map_page` corresponds to endpoint ID 2 concerning SPTM calls to the XNU bootstrap table. We can therefore easily find the `map_page` function in the SPTM binary.

Unparameterized Memory Frame Typecheck A key security check is the validation of the SPTM type of the memory frame to map to via the new page table entry. The excerpt can be seen in Listing 5.8.

```
1  if ((0x4fffffd1c0fe177eU >> (frame_sptm_type & 0x3f) & 1) != 0) {
2    DEBUG();
3  }
```

Listing 5.8: SPTM type validity checking for frame to be mapped to via `sptm_map_page`. The source code was disassembled by Ghidra.

A check is performed by right shifting the provided constant by the frame SPTM type. Every bit set in the value seen in Listing 5.8 indicates a valid mapping to memory with the corresponding frame type. Integer to type mappings can be performed via the available list of all types (see Section A.11). Valid page types that XNU is allowed to map can be seen below in Table 5.4.

Type ID	Type Name
0	SPTM_UNTYPED
7	SPTM_XNU_CODE_DBG_RW
11	XNU_DEFAULT
13	XNU_RO_DBG_RW
14	XNU_USER_EXEC
15	XNU_USER_DEBUG
16	XNU_USER_JIT
24	XNU_ROZONE
25	XNU_IO
26	XNU_PROTECTED_IO
27	XNU_COMMPAGE_RW
28	XNU_COMMPAGE_RO
29	XNU_COMMPAGE_RX
33	XNU_KERNEL_RESTRICTED
34	XNU_RESERVED_1
35	XNU_RESERVED_2
37	XNU_RESTRICTED_IO_TELEMETRY
60	SK_SHARED_RO
61	SK_SHARED_RW
63	UNKNOWN_TYPE

Table 5.4: SPTM frame types that can be mapped to via a page table entry from XNU.

As can be seen, XNU is only entitled to map a page to a small subsection of all SPTM frame types. Notably, it is not allowed to create mappings to any `ROOT_TABLE` or `PAGE_TABLE` frame types, even for those belonging to the `XNU_DOMAIN`. Such frame types are deemed security-relevant and can only be altered by SPTM. It is, however, able to map into `SK_SHARED_RO` and `SK_SHARED_RW` typed memory frames, which allows SK to share data with XNU. More details regarding SK will be provided in Chapter 6.

Target Table Typechecking A second security mechanism implemented is a conditional check based on the SPTM frame type of the table; the page table entry is supposed to be inserted. We know all possible types that the page table can be inserted into from its previous retrieval in the function `table_acquire`. The valid SPTM frame types for the table to insert the PTE into are listed in Table 5.5.

Type ID	Type Name
8	SPTM_KERNEL_ROOT_TABLE
9	SPTM_PAGE_TABLE
17	XNU_USER_ROOT_TABLE
18	XNU_SHARED_ROOT_TABLE
19	XNU_PAGE_TABLE
20	XNU_PAGE_TABLE_SHARED
21	XNU_PAGE_TABLE_ROZONE
22	XNU_PAGE_TABLE_COMMPAGE
31	XNU_STAGE2_ROOT_TABLE
32	XNU_STAGE2_PAGE_TABLE

Table 5.5: Valid SPTM frame types for tables to insert PTE into from XNU.

The performed validation ensures that XNU can only map memory using page and root tables that belong to its domain, thereby creating a full memory separation between different SPTM domains.

Whilst the previously shown Listing 5.8 performed an unconditional check for generally allowed types to map to, SPTM further proceeds to lock down the ability to map to memory frames. This is done based on the SPTM frame type of the table to which it is being inserted. Only specific frame types may be inserted into specific pages or root tables. This is achieved by retrieving a value from the global SPTM retyping structure, indexed by the SPTM type of the table to be inserted into. The check can be seen in Listing 5.9.

```
1 if ((*(ulong *)(&REMAP_STRUCTURE + (ulong)table_sptm_type * 0x60) >>
   ↪  (frame_sptm_type & 0x3f) & 1
2       ) == 0) {
3         DEBUG();
4   }
```

Listing 5.9: SPTM check for mapping to a specific frame with frame type `frame_sptm_type` via a table of frame type `table_sptm_type`. The source code was disassembled by Ghidra.

Based on the `table_sptm_type`, the global `RETYPE_STRUCTURE` is indexed and a value retrieved that encodes valid SPTM frame types to be mapped from the specific table type. The complete list of frame types that can be mapped from specific tables is included in Section A.14. The results are, in general, as expected. Root tables can map to the corresponding page tables. SPTM page tables can map to any other memory frame, regardless of its type, which appears logical given that SPTM is the page table management component running at the highest privilege level.

Further Validations We find the `map_page` function to perform multiple additional validations based on the frame to map to, the frame type of the table to insert into, and other conditional checks. New PTEs are validated, among other things, concerning their SPRR index; we were not yet able to reverse engineer. Allowed SPRR indexes appear to be defined based on the SPTM type of the table to insert into. We cannot provide a conclusive understanding of these mechanisms at this time, as we would require a current and up-to-date mapping of SPRR indices for this. However, we assume this to enforce

specific access and execution permissions for specific frame types. Exemplary, dedicated read-only page types should not be made writable via SPRR indices.

We find the `map_page` SPTM function to perform a variety of further validity checks, with further underlying rule sets governing these mappings.

5.9 SPTM Page Mapping – Conclusion

SPTM is the sole entity capable of altering page table data and introducing new virtual-to-physical memory mappings. It handles memory-mapping requests on behalf of XNU, based on a rigid set of rules regarding allowed mappings. All XNU functions altering page table data do so via an invocation of SPTM. XNU can only map to memory frames with a significantly reduced subset of SPTM types. This limits its ability to access security-relevant or isolated memory. SPTM further ensures that SPRR indices set for page table entries to be inserted into page tables also adhere to a predetermined set of rules, prohibiting XNU from arbitrarily setting SPRR indices and, consequently, access and execution rights to the memory it maps.

This restriction of memory-mapping ability via XNU essentially introduces a separation of trust throughout the system. XNU is no longer able to access nearly all parts of the system; instead, specific vulnerable system components can be scoped away in different SPTM domains, thereby limiting XNU's ability to access them. Such an architectural shift aims to protect against the dangers of a full kernel compromise in a monolithic kernel design, which typically implies a complete loss of control over all system functionality.

6 Secure Kernel

6.1 Fundamentals

The Secure Kernel (SK) is a GXF component running in GL1. It is part of the *Exclavecore* (see Chapter 7). From our previous analysis of SPTM, we know SK to have its own SPTM domain `SPTM_DOMAIN_SK`. There are four SK-specific SPTM frame types, namely `SK_DEFAULT`, `SK_SHARED_RO`, `SK_SHARED_RW` and `SK_IO`, which SK operates on.

We find entering the `SK_DOMAIN` to be made possible via SPTM through two dispatch functions registered with SPTM by SK. These functions are invoked when SPTM clients (e.g., XNU) try to call into `SK_DOMAIN`. Entering SK from XNU via `exclaves_enter` ultimately resumes execution in GL0 via an `ERET` call. SK appears to handle requests from GL0 components. Core requests include SPTM frame retyping to XNU and SK shared frame types, indicating that GL0 clients share data with XNU via this mechanism. SK appears to be the privileged call dispatching and management component in GXF, with SPTM acting as the overhead hypervisor for memory management.

6.2 SPTM Calls

We have previously discovered SPTM calls from SK in Section 5.3.2. Mapping them based on the known SPTM dispatch structure provides the results seen in Table 6.1. As all calls from SK are directed towards the `SPTM_DOMAIN`, we redact this field for readability.

Dispatch Table	Endpoint ID	SPTM Function
`SK_BOOTSTRAP`	0	`register_dispatch_table`
`SK_BOOTSTRAP`	1	`retype`
`SK_BOOTSTRAP`	2	`get_frame_type`
`SK_BOOTSTRAP`	3	Not statically provided
`T8110_DART_SK`	0	`sptm_t8110_dart_map_table`
`T8110_DART_SK`	1	`sptm_t8110_dart_unmap_table`
`RETURN_TO_CALLER`	-	Control Function
`PANIC`	-	Control Function
`EXCEPTION_STATE_SAVED`	-	Control Function

Table 6.1: SPTM calls performed from SK. Based on the assumption that the `T8110_DART_SK` dispatch table is equal to the `T8110_DART` dispatch table.

As can be seen, SK performs dispatch table registrations, which we will look into more closely in Section 6.3. It can also retype physical frames and retrieve frame types using its `SK_BOOTSTRAP` dispatch table in SPTM. Regarding the dispatch table, we were unable to find function endpoint 3 in SPTM. This might indicate it is set dynamically but it still could also be caused by issues in the decompiler.

With regards to the `T8110_DART_SK` dispatch table, SK performs table mapping and unmapping via SPTM. SK further utilizes the SPTM-provided control functions for panicking, saving exception states, and returning to callers.

Due to the complete lack of available symbols in the SK binary and other general information on its workings, and to stay within the scope of this work, we have decided to significantly limit our analysis of SK's inner workings to what we deem to be key aspects of it. This key aspect is frame retyping through SPTM, which is examined in Section 6.2. Our goal is not to understand every individual SPTM function call, but to provide an idea of the underlying concepts and their impacts on system security.

SPTM Calls – Frame Retyping

We find a variety of functions performing SPTM frame type remapping in SK. A list of all invocations is shown in Table 6.2.

Current Frame Type	New Frame Type	Invocation
SK_DEFAULT	SK_SHARED_RO	retype_to_shared_ro
-	SK_SHARED_RO	retype_to_shared1
-	SK_SHARED_RW	retype_to_shared1
-	SK_DEFAULT	map_to_sk_default1
SK_DEFAULT	SK_SHARED_RO	retype_to_shared2
SK_DEFAULT	SK_SHARED_RW	retype_to_shared2
-	XNU_DEFAULT	retype_to_XNU_default
-	SK_DEFAULT	retype_to_sk_default2

Table 6.2: Found invocations of SPTM frame retyping from SK with the corresponding function identifier. Unspecified types imply that they are dynamically retrieved from the frame to retype.

While unspecified current frame types indicate that the frame type is retrieved dynamically before the retyping operation is performed, it is essential to note that SK must adhere to the SPTM-provided rule set of allowed retyping (see Table A.11). We find SK to perform frame retyping mappings from SK_DEFAULT to SK_SHARED_RO or SK_SHARED_RW on multiple occasions. An exemplary invocation of retyping can be seen in Listing 6.1.

```
bool retype_to_shared2(undefined8 physical_address,int
 ↪  rw_ro_conditional)
{
  int current_frame_type;
  undefined4 new_type;

  current_frame_type = sptm_get_frame_type();
  if (current_frame_type == SK_DEFAULT) {
    new_type = SK_SHARED_RO;
    if (rw_ro_conditional != 0) {
      new_type = SK_SHARED_RW;
    }
    retype(physical_address,SK_DEFAULT,new_type,0);
  }
  return current_frame_type == SK_DEFAULT;
}
```

Listing 6.1: `retype_to_shared2` function in SK mapping a frame based on a provided physical address to `SK_SHARED_RW` or `SK_SHARED_RO` based on an input parameter. Only performs mapping if the underlying frame for the address is of type `SK_DEFAULT`. The source code was disassembled by Ghidra.

Recalling Table 5.4, we know XNU can map pages into memory of type `SK_SHARED_RO` and `SK_SHARED_RW`, which leads us to determine that this function makes SK-owned physical memory available to XNU. SK is responsible for and able to map memory in its domain, which is usually gapped from the kernel available to XNU if it is required for the task at hand.

6.3 SVC Handling

As SK is running in GL1, we find that it registers exception handlers for exceptions taken from GLo components. As mentioned previously, multiple components running in GL issue SVCs to the secure kernel. The handling of such calls will be looked at in this section.

Similar to our analysis of SPTM exception handlers, we can examine writes to the `VBAR_GL1` register in SK and identify various writes to it. Next to the actual exception handler, we also find an early entry exception handler registration, which acts similarly to the SPTM one. The actual exception handler base is registered at `ExceptionHandlerBase`. The function performing the exception handling for synchronous exceptions from lower levels can be found at offset `+0x400` [18]. Following this path shows a wrapper method, and finally the actual handler `SVC_handler_from_lower_EL`.

We refrain from performing a full analysis on this function, but rather look at it conceptually. SK handles SVCs with immediate values from 0 to 5. For other values, it appears to perform a context restore and returns to the caller via `RestoreContextAndEret`. For SVC #0 calls, we find the following implementation.

```
1     if (svc_imm == 0) {
2       allinged_pointer_1 =
3            (ulong *)(pointer_1 & 0b00000000000000000000000011111111
              ↪  11111111111111111111000000)
4       ;
5       if (allinged_pointer_1 != (ulong *)0x0) {
6         (*(code *)(&FunctionTable)[(*allinged_pointer_1 >> 0x3a) *
           ↪  0xb])
7                   (allinged_pointer_1,pointer_2);
8         goto PrepareContextAndEret;
9       }
```

Listing 6.2: Request dispatching in `SVC_handler_from_lower_EL` in SK, for SVC call with IMM value 0. A function is indexed in a function table based on data pointed to by a pointer provided by the SVC call. The source code was disassembled by Ghidra.

An input pointer provided from context at exception call is aligned by setting its lower 6 bits to zero. Furthermore, its uppermost 32 bits are also set to zero. This ensures the pointer is within a specific range and aligned to a structure. The fact that the upper 32 bits of the input pointer are not considered indicates that it is a userspace pointer from the GLo component calling into SK[9] SK then retrieves the value pointed at by the updated pointer and performs a right shift operation on it. The result of said shift is used as an index into `FunctionTable`. Essentially, we assume SK retrieves user-provided data to index an internal function table, serving the client's request. The control flow of such a retrieval can be seen in Figure 6.1.

[9]We can assume this due to the standard memory layout of ARM, where we find user space virtual addresses usually assigned to the lower memory regions [9].

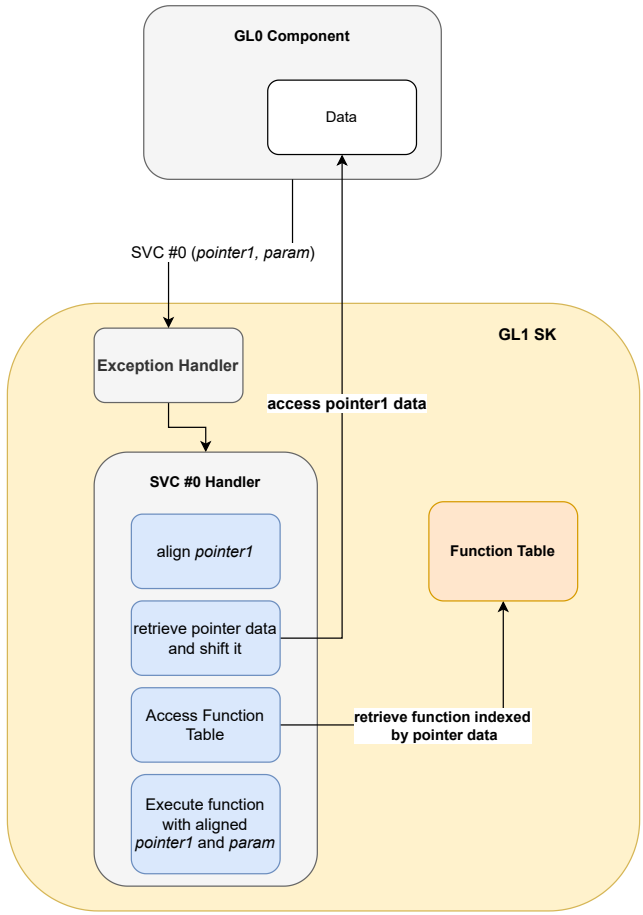


Figure 6.1: Assumed SVC #0 handling by SK. A client-provided pointer is aligned and used to access GL0 user space data. Based on the data retrieved, an internal function table is accessed and the function executed with two client-provided arguments.

We are not provided with any table limits for the function table accessed. Looking through the table entries, we find function calls that call the previously discovered retyping function (Table 6.2). From this, we know that GL0 clients of SK call into it for the retyping of memory frames they are using. Such retyping may be performed to make memory available to XNU (retyping to `SHARED` types, or to acquire new SK memory frames (retyping to `SK_DEFAULT`).

The SVC handling for different `IMM` values and the further functionality offered by SK to its GL0 clients has not yet been reverse-engineered by us and is left for future work. Based on the seen function table, SK appears to offer a wide range of functions; however, with the lack of any symbols, reverse engineering these would go beyond the scope of this work.

Dispatch Table Registration

We have previously seen that XNU calls into the SK domain through STPM (see Section 5.3.1). These calls from XNU functions `exclaves_enter` and `exclaves_bootinfo` are dispatched by SPTM to SK. However, dispatch tables for `SK_DOMAIN` are found not to

be statically included in SPTM, but are instead dynamically registered at runtime by SK. This is done via `sptm_register_dispatch_table` calls, which are performed by SK (see Table 6.1). The two register calls performed can be seen in Listing 6.3.

```
sptm_register_dispatch_table(0,&SPTM_dispatch_function_0,2);
sptm_register_dispatch_table(1,&SPTM_dispatch_function_1,1);
```

Listing 6.3: Calls to `sptm_register_dispatch_table` from SK function `FUN_ffffff800000aecc`. SK registers two dispatch functions that allow calling into SK when calling the domain `SK_DOMAIN`. The source code was disassembled by Ghidra.

The function calling these registrations is called nested from the SK entry function, by which we know SK makes itself available to SPTM soon after boot. The two functions SK registers as dispatch functions are `SPTM_dispatch_function_0` and `SPTM_dispatch_function_1`. As known from our previous reverse engineering of the SPTM dispatch table registration process (see Section 5.3.5), the third parameter of the registration call is the bitwise-encoded permission. A set bit in the value indicates that a domain is allowed to call into the corresponding dispatch table. We therefore know that `SPTM_dispatch_function_1` is only callable by `XNU_DOMAIN`, and `SPTM_dispatch_function_2` is only callable by `SPTM_DOMAIN`. The dispatch registration performed by SK is shown in Figure 6.2.

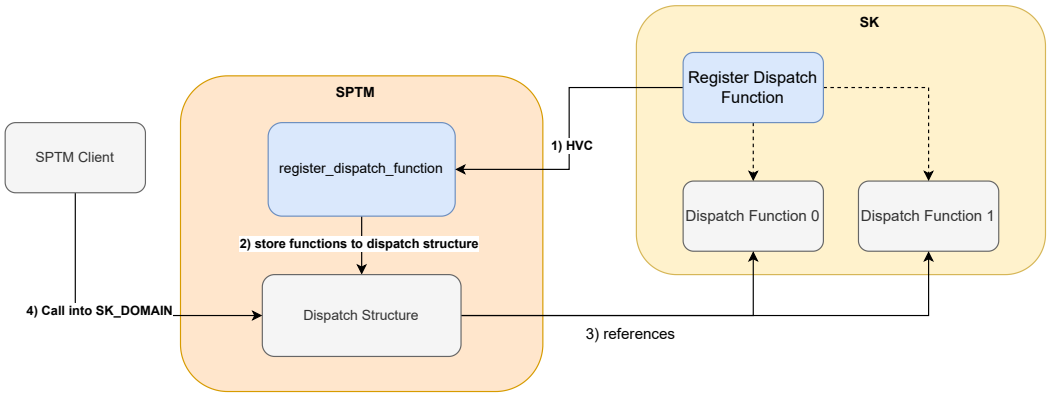


Figure 6.2: Process of SK dispatch function registration in SPTM.

Looking at `SPTM_dispatch_function_0`, we find a call to a function we named `SK_called_from_sptm_dispatch_function_0`. This function performs the actual dispatching via further method calls to `dispatch_0_handle_endpoints`. It checks whether the provided endpoint is either 0 or 1 (recall 0 for `EXCLAVES_ENTER`, 1 for `EXCLAVES_BOOTINFO`). For the bootinfo endpoint call, a return to the caller is performed via the specific SPTM call. For the Exclaves enter endpoint, we find a nested method call to `RestoreContextAndEret`.

```
undefined1  [16] RestoreContextAndEret(void)

{
  undefined8 *puVar1;
  undefined1 (*saved_registers) [16];

  puVar1 = (undefined8 *)TPIDR_GL1;
  saved_registers = (undefined1 (*) [16])*puVar1;
  if (*(long *)(saved_registers[0x13] + 8) == 0) {
    tpidr_el0 = *(undefined8 *)saved_registers[0x12];
    tpidrro_el0 = *(undefined8 *)(saved_registers[0x12] + 8);
    fpcr = *(undefined8 *)(saved_registers[0x11] + 8);
    fpsr = *(undefined8 *)saved_registers[0x11];
    SPSR_GL1 = *(undefined8 *) (saved_registers[0x10] + 8));
    sp_el0 = *(undefined8 *)saved_registers[0x10];
    ELR_GL1 = *(undefined8 *)(saved_registers[0xf] + 8);
    ExceptionReturn();
    return *saved_registers;
  }
  *(undefined8 *)(saved_registers[0x13] + 8) = 0;
  tpidr_el0 = *(undefined8 *)saved_registers[0x12];
  tpidrro_el0 = *(undefined8 *)(saved_registers[0x12] + 8);
  fpcr = *(undefined8 *)(saved_registers[0x11] + 8);
  fpsr = *(undefined8 *)saved_registers[0x11];
  SPSR_GL1 = *(undefined8 *)(saved_registers[0x10] + 8);
  sp_el0 = *(undefined8 *)saved_registers[0x10];
  ELR_GL1 = *(undefined8 *)(saved_registers[0xf] + 8));
  ExceptionReturn();
  return *saved_registers;
}}
```

Listing 6.4: `RestoredContextAndEret` function in SK. The source code was disassembled by Ghidra.

We find what appears to be a context restore and preparation. The GL1 `ELR` register is written to a value retrieved from the thread identifying information register (`TPIDR_GL1`). Finally, both cases perform an `ExceptionReturn()` (`ERET` [5]) call, which returns to the address previously set to the `ELR`.

We deduce that whilst SK acts as SPTM call taker, it delegates actual request handling to GL0 components it returns to via `ERET`. We cannot confirm where execution is continued after the instruction, but assume it to be `xnuproxy` (contained in the `sharedcache` binary; more on `xnuproxy` in Section 7.9.5).

7 Exclaves

Exclaves are found to contain specific subsets of sensitive resources, which are scoped in the `SK_DOMAIN`, and by that naturally isolated from XNU. We find them to offer services that clients can call via exposed interfaces. Exclaves offer a variety of different communication mechanisms, including a dedicated IPC framework called Tightbeam. They are used for further compartmentalization and division of trust away from XNU and towards a more microkernel-style architecture.

7.1 Exclave System Components

Exclaves encompass a range of components, which we will list briefly here.

7.1.1 Exclavecore

A large part of the Apple Exclave ecosystem is the *Exclavecore*. It consists of the SK found running in GL1, and a variety of other system binaries we determine to run in GL0. The GL0 binaries are the following:

- ExclaveStackshotServer
- pmm_exclave
- roottask
- scheduler
- sharedcache

Except for sharedcache in the context of our analysis of `xnuproxy` (see Section 7.9.5), we have not further analyzed any of these binaries so far. We do, however, find them all to perform SVCs, which we assume to be handled by SK as described in Chapter 6. GL0 components in the Exclavecore appear to be subject to significant changes between different firmware versions. We assume the reason is that Exclaves are a relatively new security mitigation still under constant development.

The Exclavecore bundle is located at `Firmware/image4/exclavecore.<version>.RELEASE.im4p` in the firmware IPSW file. It can be extracted using the `ipsw` tool with the command `ipsw extract -x *.ipsw`.

7.1.2 ExclaveKit

On devices supporting Exclaves, a new Apple Disk Image (DMG) is found. We find this to contain the `ExclaveKit`. The `ExclaveKit` contains a variety of `Frameworks` and `PrivateFrameworks`. Frameworks are dynamic libraries, with `Frameworks` being usable from App Store Apps, whilst `PrivateFrameworks` are meant to be only used by Apple's apps [51]. A complete listing of the frameworks discovered in the `ExclaveKit` DMG can be found in Section A.15. We assume the `ExclaveKit` to store the actual Exclave's code. Among others, we find the private `Tightbeam.framework`, which will be a focus of our analysis in Section 7.10.

7.2 Exclave Memory Fundamentals

Although it may appear premature, examining Exclave memory fundamentals provides important context for the upcoming analysis. `osfmk/arm/machine_routines.h` lists a `ml_paddr_is_exclaves_owned` function, which, for a provided physical address, verifies whether it is "owned by the secure world". From the underlying implementation we find this is the case for frame type `SK_DEFAULT` and `SK_IO`. This verifies that Exclaves are running in the secure world, which appears to denote the `SK_DOMAIN`. Only the two named types are exclusively used by the secure world. `SK_SHARED_RW` and `SK_SHARED_RO` frames are not "exclusively exclaves frames", but instead can be shared with XNU (see `osfmk/arm64/machine-routines.c`, l. 2955). This confirms our previous analysis results in Table 5.4, which shows that XNU is allowed to map memory to frames of the named types, which is a requirement for memory access. This concept is depicted in Figure 7.1.

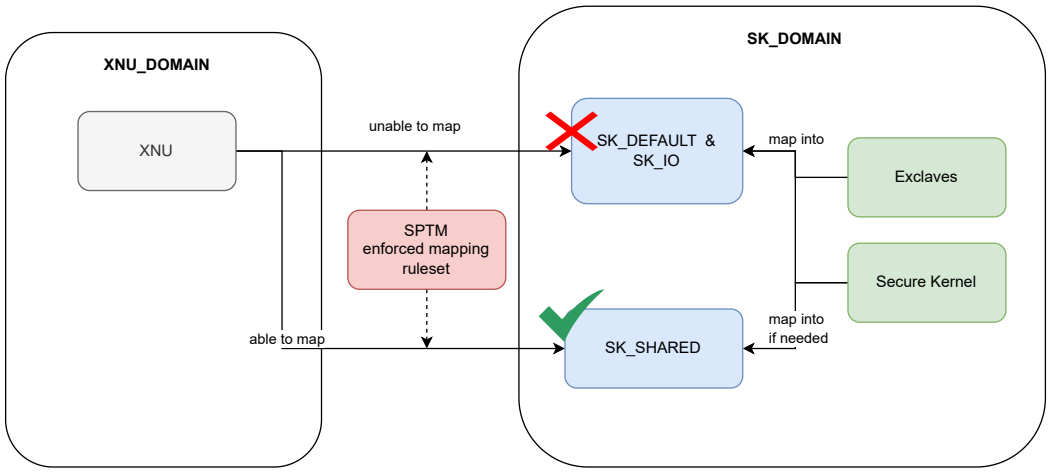


Figure 7.1: Exclaves run in the `SK_DOMAIN`, mapping memory into `SK_DEFAULT` and `SK_IO` type frames for execution separated from XNU. If the need arises for information exchange with XNU, Exclaves may also map into shared frame types.

7.3 Resources

The open-source code parts of XNU offer some key insights into resources within Exclaves (`osfmk/kern/exclaves_resource.h`). According to comments in the source code, "exclaves provide a fixed static set of resources available to XNU". Types of resources include, among others, Conclave managers, services, sensors, and buffers. As noted in the comment, Exclave resources appear to be static at runtime, being enumerated once during boot for setup, and not being alterable after. This also entails that if resources gap specific capabilities from other parts of the XNU kernel, we can not alter this past boot and must rely on what was initially set up. Furthermore, if Exclave resources are gapped from XNU but made available, there must be mechanisms to call them, apart from standard memory mapping. We will analyze these in Section 7.9.

Resources have a specific name, a type, and an identifier, with the identifier being used for addressing between XNU and Exclaves. They are further scoped in so-called domains, which will be examined more closely in Section 7.4. The `exclaves_resource_t`

type definition as made available in `osfmk/kern/exclaves_resource.h` is shown in Listing 7.1.

```
#define EXCLAVES_RESOURCE_NAME_MAX 128
typedef struct exclaves_resource {
    char                r_name[EXCLAVES_RESOURCE_NAME_MAX];
    xnuproxy_resourcetype_s r_type;
    uint64_t            r_id;
    _Atomic uint32_t    r_usecnt;
    ipc_port_t          r_port;
    lck_mtx_t           r_mutex;
    bool                r_active;
    bool                r_connected;

    union {
        conclave_resource_t     r_conclave;
        sensor_resource_t       r_sensor;
        exclaves_notification_t r_notification;
        shared_memory_resource_t r_shared_memory;
    };
} exclaves_resource_t;
```

Listing 7.1: The `exclaves_resource_t` type as made available in `osfmk/kern/exclaves_resource.h`.

It stores general information about the resource, such as name, type, and identifier, and, more interestingly, type-specific additional information in the form of either a Conclave, notification, shared memory, or sensor resource. The specific type definitions can be found in Section A.16. We will analyze Conclaves in detail in Section 7.5. Regarding Exclave resource types, we find references to the following resource types in the XNU open-source code:

- `XNUPROXY_RESOURCE_CONCLAVE_MANAGER`
- `XNUPROXY_RESOURCE_NOTIFICATION`
- `XNUPROXY_RESOURCE_SERVICE`
- `XNUPROXY_RESOURCE_NAMED_BUFFER`
- `XNUPROXY_RESOURCE_ARBITRATED_AUDIO_BUFFER`
- `XNUPROXY_RESOURCE_SENSOR`
- `XNUPROXY_RESOURCE_SHARED_MEMORY`
- `XNUPROXY_RESOURCE_ARBITRATED_AUDIO_MEMORY`

We further find that Exclave resources appear to have a standard `ipc_port_t` indicating potential access via standard IPC mechanisms. This will be discussed in Section 7.9.

7.4 Exclave Domains

The XNU open-source code reveals a great deal about Exclaves and their structuring (`osfmk/kern/exclave_resources.c`, l.90). We know that resources "are scoped by what entities are allowed to access them". This concept of a scope is what is denoted

as `domains`. These domains are not interchangeable with SPTM domains, but rather represent a distinct concept. There appears to be a two-level table scheme indexing Exclave resources for access. A root table indexing Exclave domains, with second-level tables storing actual Exclave resources.

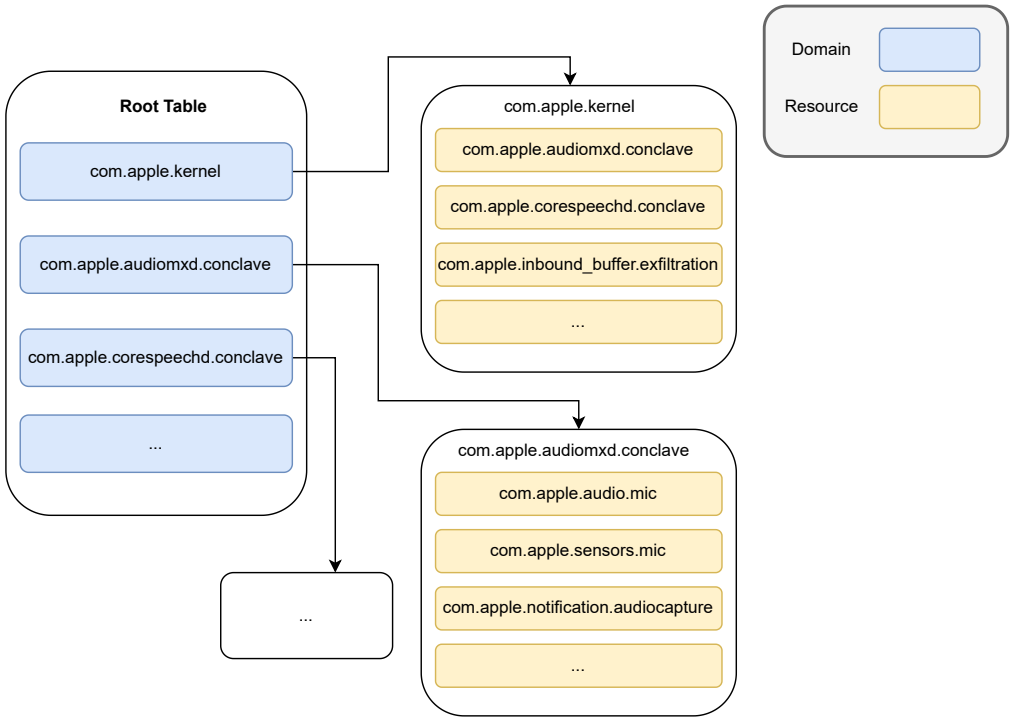


Figure 7.2: Exclaves table structure with a root table holding domain references and a pointer to the actual domains with their respective Exclave resources, adapted [32].

Resource Enumeration

If resources are predetermined at boot, we have to find them within the firmware. By searching for known resource name patterns like `com.apple.conclave.*` extracted from `osfmk/kern/exclaves_resource.c`, we can determine where in the firmware the resource list is stored. We find them in the `sharedcache` binary, a part of the `Exclavecore`, which will be looked at in more detail in our analysis of the underlying `xnuproxy` in Section 7.9.5. We can, however, already determine a `resource_registration` function, which accesses a global structure containing resource and domain pairings. The extracted list of domains with corresponding resources can be found in Section A.17. The script used to extract these resources is provided in Listing A.13. A variety of resources are found via this, including sensors, notifications, inbound buffers, and generic services. It also appears that specific resources can be correlated with multiple domains at a time, as exemplified by the `com.apple.sensors.mic` resource. We also find resource duplication, even within the same domain space, which might hint towards resources with different permissions.

Resource Discovery and Initialization Resources are discovered and initialized as part of the early Exclaves boot process. The discovery is performed by the `exclaves_resou`

rce_init function in osfmk/kern/exclaves_resource.c. The function initializes the root table and then proceeds to iterate every resource via a call to exclaves_xnu_proxy_send with command XNURPOXY_CMD_RESOURCE_INFO. We will analyze calls to xnuproxy more thoroughly in Section 7.9.5, but can infer that it is responsible for resource management.

The exclaves_resource_init function proceeds by verifying that the resource domain exists in the root table and creates it if it does not. This is a prerequisite to allocating the new resource in the domain. After this, type-specific initialization is performed.

- For **services**, this verifies that the maximum number of services per Conclave is not exceeded.
- For **notification resources**, this calls exclaves_notification_init, which does only initialize a kernel list field in the resource.
- For **Conclave manager resources**, this calls exclaves_conclave_init, which forwards the call to exclaves_conclave_launcher_init parameterized with the discovered resource ID and a connection output parameter. This function call performs what we assume to be Tightbeam connection setup on the discovered resource via conclave_launcher_conclavecontrol__init. It stores a tb_client_connection_t in the Conclaves r_control field. This is the Tightbeam connection that interacts with the Conclave. Tightbeam, as the Exclaves communication framework, will be analyzed in detail in Section 7.10. Based on this type-specific resource initialization, we determine that Conclave manager resources are the only resources to set up and hold a Tightbeam connection.

After all resources have been enumerated and type-specific initialization has been performed, populate_conclave_service is called. This function populates the service bitmaps of all discovered Conclaves for all services belonging to their respective domains. Essentially, this function registers all available resources for each Conclave manager resource, where Conclave manager resources are these Exclave resources that have a designated Conclave. Conclaves will be looked at in more detail in Section 7.5.

7.5 Conclaves

From the exclaves_resource_t and conclave_resource_t type definitions looked at in Section 7.3, we know Conclaves to be a specific subtype of resource, optionally contained within an Exclave resource. The type definition of a Conclave are shown in Listing 7.2. Conclaves themselves group multiple services in a bitmap, have an assigned task and thread, and store internal state and request information. Additionally, they include a tb_client_connection_t, a Tightbeam connection.

```
typedef struct {
    conclave_state_t        c_state;
    conclave_request_t      c_request;
    bool                    c_active_downcall;
    bool                    c_active_stopcall;
    bool                    c_active_detach;
    tb_client_connection_t  c_control;
    task_t                  XNU_PTRAUTH_SIGNED_PTR("conclave.task")
     ↪  c_task;
    thread_t                XNU_PTRAUTH_SIGNED_PTR("conclave.thread")
     ↪  c_downcall_thread;
    bitmap_t
     ↪  c_service_bitmap[BITMAP_LEN(CONCLAVE_SERVICE_MAX)];
} conclave_resource_t;
```

Listing 7.2: `conclave_resource_t` type definition from `osfmk/kern/exclaves_resource.h`.

Conclaves hold an internal state and request. They also have boolean parameters indicating what we assume to be requested transitions. The `c_control` field of type `tb_client_connection_t` holds the Tightbeam communication interface for the Conclave, and is initiated in the previously looked at `exclaves_conclave_init` function. Conclaves furthermore hold a specific task and a downcall thread. Finally, they have a service bitmap field, which is populated with services belonging to the specific Conclave during resource enumeration (Section 7.4). The services registered in this bitmap are uniquely identified by their own resource ID. From the previous Exclave resource enumeration, we know Exclave resources holding Conclaves to be of type `XNUPROXY_RESOURCE_CONCLAVE_MANAGER`. We denote the Conclave manager resource holding the Conclave as the responsible Conclave manager for that specific Conclave. The `osfmk/kern/exclaves_resource.h` header reveals that `conclave_request_t` stores a requested state transition. The mapping are depicted in Table 7.1. A request may

Value	State
0	`CONCLAVE_R_NONE`
1	`CONCLAVE_R_LAUNCH_REQUESTED`
2	`CONCLAVE_R_SUSPEND_REQUESTED`
4	`CONCLAVE_R_STOP_REQUESTED`

Table 7.1: Integer to request mapping for `conclave_request_t`.

be stored for all transitions into and out of a running Conclave, but not for the attaching and removing. The header also includes information on the Conclave lifecycle, which is discussed in Section 7.5.

We conclude that Conclaves are contained within Exclave resources of the Conclave manager type. They scope a variety of different services together, which are in turn also Exclave resources. Services are recognized as belonging to a Conclave via their domain. Every Conclave defines its own domain to which the services are assigned. The services are registered to the Conclave via a bitmap and uniquely identifiable by their own resource ID. We assume communication towards Conclaves to be performed using Tightbeam on the exposed Tightbeam connection field. The responsible Conclave manager appears to handle requests directed to the Conclave and is responsible for its management. We

further know that for each Conclave domain, the respective Conclave manager has to be in the `com.apple.kernel` domain (`osfmk/kern/exclaves_resource.c`, ll. 156f).

Conclave Lifecyle

Regarding the Conclave-internal state, we are provided with additional information in the `exclaves_resource.h` header file. An included state machine are depicted in Figure 7.3.

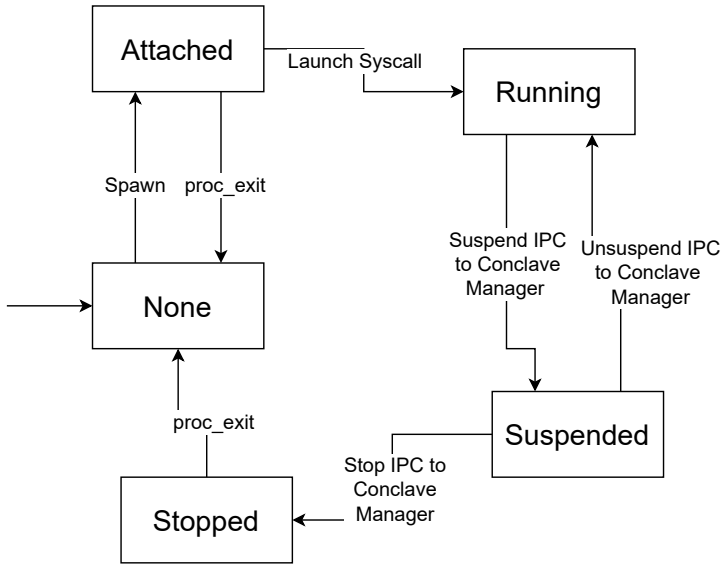


Figure 7.3: Conclave state machine adapted from the `osfmk/kern/exclaves_resource.h` header file.

Conclaves can be attached to tasks and then started up via a system call. Whilst they are running, the responsible Conclave manager can be called into via the exposed Tightbeam connection. This is not possible whilst they are suspended. Stopping the Conclave also completely terminates IPC to the Conclave manager. The respective integer-to-state mapping, as made available in the header file, can be seen below in Table 7.2.

Value	State
0	`CONCLAVE_S_NONE`
1	`CONCLAVE_S_ATTACHED`
2	`CONCLAVE_S_RUNNING`
3	`CONCLAVE_S_STOPPED`
4	`CONCLAVE_S_SUSPENDED`

Table 7.2: Integer to state mapping for `conclave_state_t`.

We know that Conclaves can be attached to Mach tasks by the existence of the task field in the Conclave type and the respective Conclave field in the task type. Conclaves are found to be attached to tasks via `exclaves_conclave_attach` (`exclaves_resource.c`, l. 1347). The function sets the specific Conclave and task fields, and the Conclave state to `CONCLAVE_S_ATTACHED`.

We find it invoked from `task_add_conclave` (`osfmk/kern/task.c` l. 9948). The function reveals an entitlement constraint for attaching Conclaves. Only `launchd` and tasks with the `com.apple.private.exclaves.conclave-spawn` entitlement are allowed to attach a Conclave to a task. Furthermore, to have a Conclave attached, a task must have either the `com.apple.private.exclaves.conclave-host` or `com.apple.private.exclaves.conclave-spawn` entitlement.

We find `task_add_conclave` to be called from `posix_spawn` in `kern_exec.c` (l. 4294), which shows that Conclaves are usually attached to a task during task spawn. The process of attaching a Conclave to a task is shown in Figure 7.4.

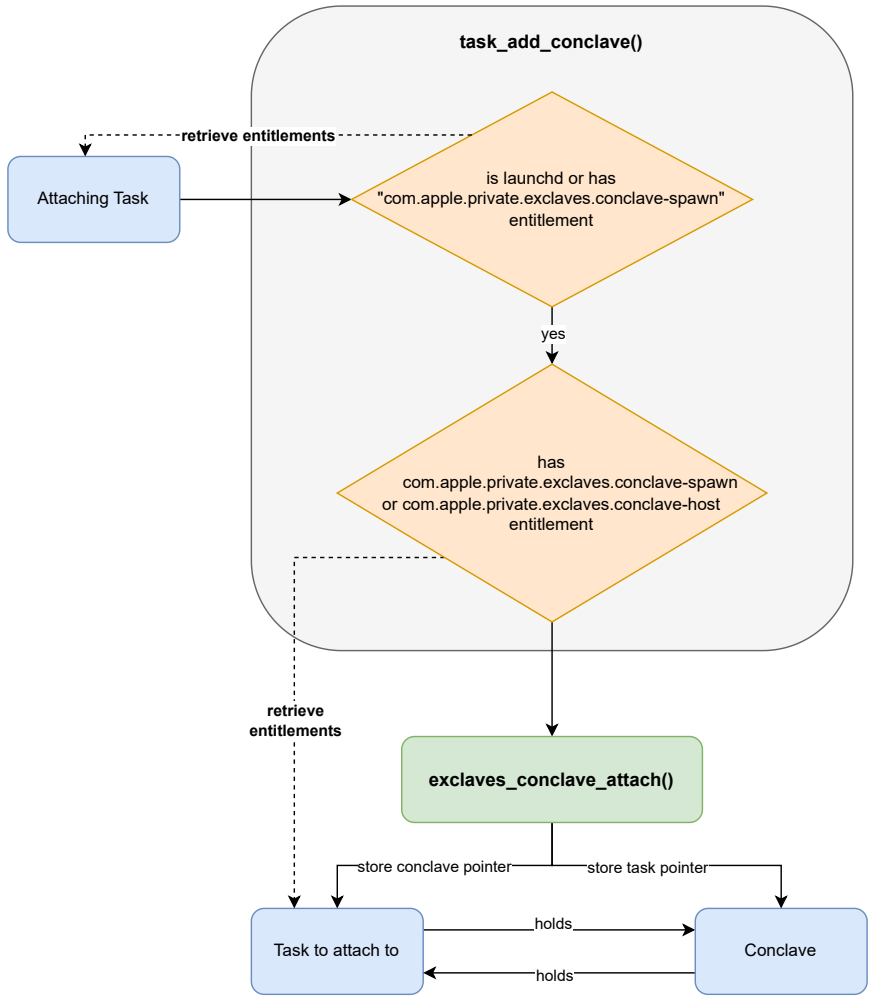


Figure 7.4: Control flow for attaching a Conclave to a task.

As we have previously found that Conclaves are attached as part of `posix_spawn`, it is logical that we find the Conclave ID to be a Portable Operating System Interface (POSIX) spawn argument (`bsd/sys/spawn_internal.h`, l. 539). We can therefore examine launch property lists for tasks we know to have Conclaves attached (e.g., `audiomxd`) and,

in fact, find a `_Conclave` key. The entry in the `audiomxd` launch property list[10] can be seen in Listing 7.3, and shows the corresponding Conclave identifier.

```
1   "_Conclave" => "com.apple.audiomxd.conclave"
```

Listing 7.3: Conclave ID entry in launch property list of `audiomxd`.

Theoretically, it is also possible to attach Conclaves to tasks by directly calling `task_add_conclave` from an appropriately entitled task. However, we were unable to observe any such invocation. We therefore assume that Conclave attachment is solely done by launchd when spawning a task. Therefore, any task that requires an attached Conclaves needs to have the previously looked at key in its launch property list.

7.6 Exclaves Scoped Functionality

From the retrieved resource list (see Section 7.4), we can infer that Exclaves mainly scope functionality regarding sensor access and managing audio and video recording. This entails a variety of different resources and domains, ranging from sensor resources specific to audio drivers, and the relevant buffers to allow for copying out sensor data to XNU. We furthermore find that Apple scopes Apple Neural Engine (ANE) related functionality in Exclaves. Resources are subject to change, just like the rest of the Exclaves ecosystem. The amount of resources is increasing with newer firmware versions.

7.7 Userspace Exclaves Functions

As userspace components need access to Exclaves to invoke operations on them, there exist userspace wrappers to allow for this. We can find these wrapper functions in the open source code XNU files in `libsyscall/wrappers/exclaves.c`. In the XNU source code, the libsyscall wrapper published a total of 19 functions. The total list, including function signatures, can be found in Listing A.12. The following functions are included:

- `exclaves_endpoint_call`
- `exclaves_outbound_buffer_[create/copyout]`
- `exclaves_inbound_buffer_[create/copyin]`
- `exclaves_named_buffer_[create/copyin/copyout]`
- `exclaves_launch_conclave`
- `exclaves_lookup_service`
- `exclaves_boot`
- `exclaves_audio_buffer_[create/copyout/copyout]`
- `exclaves_sensor_[create/start/stop/status]`
- `exclaves_notification_create`

[10] Located at `/System/Library/LaunchDaemons/com.apple.audiomxd.plist`.

We find most of these functions implemented in the `libsystem_kernel.dylib` of the firmware's Dynamic Linker (DYLD) shared cache. In the analyzed firmware, the `exclaves_named_buffer_[create/copyin/copyout]` functions have not yet been fully implemented, but instead return the error value `0x46`, which maps to *"(os/kern) service not supported"*. We assume this is, at least with regard to userspace invocation, unimplemented functionality that will be released in the future.

All implemented functions perform the actual invocation via the `exclaves_ctl_trap` Mach trap. Analyzing this function in Ghidra shows the invocation of Mach trap `-88`. A negative number is expected for a Mach trap, in contrast to standard POSIX calls with positive numeration [29].

7.7.1 Handling of `exclaves_ctl_trap`

This section focuses on the kernel-side handling of `exclaves_ctl_trap` invocations by user-space components. The `exclaves_ctl_trap` handler is found at `0xfffffff00813a220` in the iOS 18.4 kernelcache, and in the `osfmk/kern/exclaves.c` file in *xnu-11215*. The handling of the `exclaves_boot` is done separately from all other function invocations, as it requires different permissions, because tasks can not yet be associated with Conclaves.

Function Invocation Requirements Invoking Exclaves functionality (except for Exclave boot) has a very specific set of requirements. It can only be performed by "properly entitled tasks which can operate in the kernel domain, or those which have joined conclaves" (`osfmk/kern/exclaves.c`).

Verification on whether a task is allowed to operate in the kernel domain is performed by checking the `com.apple.private.exclaves.kernel-domain` entitlement. The kernel itself may also operate in the kernel domain. As another prerequisite for function execution, the execution waits until `Exclavecore` has successfully booted.

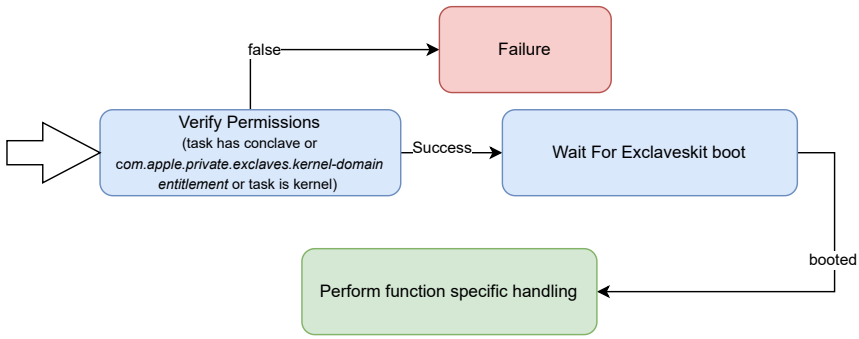


Figure 7.5: Process of checking proper caller entitlement for an Exclave functionality user space invocation.

After performing the initial handling verification, the actual operations are performed with further operation-specific parameter verification. As we know, Conclaves are only attached to tasks based on a static set of resources and the correct boot arguments; the list of user-space components able to invoke Exclave functionality is strictly limited to those.

Notes on macOS While this paper focuses on Apple's mobile operating systems, some interesting findings regarding Exclave functionality in macOS came to light during an

attempt to test functionality invocation on an M4 Mac mini. Trying to call any Exclave function from a user-space process context always resulted in a return of *"(os/kern) service not supported"*. An analysis of macOS 15.4.1 revealed that the wrapper functions and the `exclaves_ctl_trap` are indeed implemented. However, they seem unsupported on the kernel side as of now, as the XNU Mach control trap handler for the relevant trap is not implemented and instead returns the relevant error code.[11] It appears that Exclaves are only supported on iOS/iPadOS. This might change in the future.

7.7.2 Userspace Exclave Interaction – Practical Experiment on SRD

In an attempt to display our ability to interact with Exclaves from userspace, we perform an experiment in which we inject the `corespeechd` with a `dylib` that invokes Exclaves functionality. We choose `corespeechd` as a target for this attempt because we know it to have an attached Conclave, and it remains robust even when tampered with. Unlike other candidates, `corespeechd` is stable and not regularly crashes (e.g., `audiomxd`). We can trace its control flow regarding Exclaves partially. As part of our preparations, we trace invocations of `exclave_ctl_trap` on the SRD, and try to recreate them. An excerpt from the logs retrieved during dynamic reverse engineering of the SRD can be found in Section A.30 and A.31, which we provide as the first public look into live logs from Exclave-supported devices.

We successfully create a sensor resource and an audio buffer, receive a return port for both calls, and start the sensor resource; however, copying out of the audio buffer is not successful. Even though the access is permitted and we get successful return codes from the kernel, all bytes returned into the audio buffer are Null.

Furthermore, despite actively accessing an audio buffer, no Secure Indicator Light (SIL) appears. When manually tracing for Secure Indicator Light (SIL) rendering in lldb while triggering the SIL via on-board applications on iOS, parts of the rendering appears to be done by the userspace process. This contradicts our expectation that SIL rendering should take place and be monitored in GL.

We are unsure why our setup fails, but we assume it is related to the buffer registration for the sensor to write to. Nonetheless, it can still be considered a success that we were able to create Exclaves resources and invoke functions on them from userspace from an appropriately entitled task. The log from the successful creation of the sensor and buffer via our custom `dylib` and sensor start is contained in Listing 7.4.

```
1 Jul  1 15:35:05 corespeechd(libExclaves5.A.dylib)[446] <Notice>: Sensor
   ↪ created: port=0x1003
2 Jul  1 15:35:05 corespeechd(libExclaves5.A.dylib)[446] <Notice>: Buffer
   ↪ created: port=0x1103
3 Jul  1 15:35:05 corespeechd(libExclaves5.A.dylib)[446] <Notice>:
   ↪ Allocated local memory
4 Jul  1 15:35:05 corespeechd(libExclaves5.A.dylib)[446] <Notice>:
   ↪ exclaves_sensor_status polled value: 0x1
5 Jul  1 15:35:06 corespeechd(libExclaves5.A.dylib)[446] <Notice>: Sensor
   ↪ started successfully.
```

Listing 7.4: Logs written from our custom dylib `libExclaves5.A.dylib` injected into corespeechd, invoking Exclaves functionalities from userspace.

[11] `exclaves_ctl_trap` is located at address `0xfffffe00087bdbf0` in `kernelcache.release.Mac16_5_6_7_8_9_11` in macOS 15.4.1 (24E263).

The source code for the dylib used for this experiment can be found on GitHub.[12] It has been based on Apple's guide on creating dynamic libraries [2].

7.8 Kernel Entry Point

In the XNU open source code, we find the Exclave entry point for IPC calls from the XNU kernel. The function `exclaves_endpoint_call` is declared in `osfmk/kern/exclaves.c`, and shown in Listing 7.5.

```
#pragma mark kernel entry points

kern_return_t
exclaves_endpoint_call(ipc_port_t port, exclaves_id_t endpoint_id,
    exclaves_tag_t *tag, exclaves_error_t *error)
{
#if CONFIG_EXCLAVES
    kern_return_t kr = KERN_SUCCESS;
    assert(port == IPC_PORT_NULL);

    Exclaves_L4_IpcBuffer_t *ipcb = Exclaves_L4_IpcBuffer();
    assert(ipcb != NULL);

    exclaves_debug_printf(show_progress,
        "exclaves: endpoint call:\tendpoint id %lld tag 0x%llx\n",
        endpoint_id, *tag);

    ipcb->mr[Exclaves_L4_Ipc_Mr_Tag] = *tag;
    kr = exclaves_endpoint_call_internal(port, endpoint_id);
    *tag = ipcb->mr[Exclaves_L4_Ipc_Mr_Tag];
    *error = XNUPROXY_CR_RETVAL(ipcb);

    exclaves_debug_printf(show_progress,
        "exclaves: endpoint call return:\tendpoint id %lld tag 0x%llx "
        "error 0x%llx\n", endpoint_id, *tag, *error);

    return kr;
#else /* CONFIG_EXCLAVES */
#pragma unused(port, endpoint_id, tag, error)
    return KERN_NOT_SUPPORTED;
#endif /* CONFIG_EXCLAVES */
}
```

Listing 7.5: The `exclaves_endpoint_call` function, which is kernel Exclaves entrypoint, from `osfmk/kern/exclaves.c`.

The function is found to perform a call into Exclaves using `xnuproxy`. Our analysis further shows that it is called in the context of Tightbeam messaging. We will analyze this in Section 7.10.

[12] https://github.com/Moritz209/exclaves_dylib

7.9 Calling into Exclaves

For the XNU open-source code alone, it becomes obvious that calling and scheduling into Exclaves is a notoriously complex task. It appears to be performed in various ways, some of which we will analyze in the following section. Our analysis here has no claim to completeness, and due to the significant parts of the redacted source code that we have to reverse engineer, we also cannot claim correctness.

7.9.1 Thread Scheduling

We find that threads are scheduled into Exclaves when performing endpoint calls. This is confirmed by the inclusion of an interrupt state flag value in `thread.h` indicating current thread execution in SK or Exclaves userspace. The flag value definition can be seen below in Listing 7.6.

```
__options_decl(thread_exclaves_intstate_flags_t, uint32_t, {
    /* Thread is currently executing in secure kernel or exclaves
     ↪  userspace
     * or was interrupted/preempted while doing so. */
    TH_EXCLAVES_EXECUTION                   = 0x1,
});
```

Listing 7.6: Thread interrupt state value definition for execution in Secure Kernel or Exclaves userspace, in `osfmk/kern/thread.h`.

We find that this value is set before performing `sk_enter` and unset afterward. From this, we know that `sk_enter` performs the context switch via our known call to SPTM. This appears to be realized via scheduling context IDs, which we find are managed by xnuproxy. Threads get scheduled into the secure world in so-called downcalls, and may perform upcalls back to XNU when requesting XNU services during downcall handling [21]. In this work, we did not focus on the scheduling process itself, but instead on the underlying communication mechanisms. Background on the Exclaves scheduling process can be found in previous work [42].

7.9.2 Exclave thread execution states

We can draw information regarding the different calls into Exclaves from `osfmk/kern/thread.h`, which lists `thread_exclaves_state_flags_t`, the different values it may hold, and provides additional context. We find this value to be consistently altered in `osfmk/kern/exclaves.c`, tracking the current execution status of the thread. A listing of the available values, their respective bit in the flag, and context provided in the header file can be seen in Table 7.3.

Flag Value Name	Flag Value	Context
`TH_EXCLAVES_RPC`	0x1	Thread is currently in RPC handling from either XNU or user space, but it may currently run in XNU whilst performing an upcall. Reentering Exclaves or returning to user space is not permitted.

Continued on next page

Table 7.3 – continued from previous page

Flag Value Name	Flag Value	Context
TH_EXCLAVES_UPCALL	0x2	During RPC handling, the thread made an upcall RPC request back into XNU. Reentering Exclaves or returning to user space is not permitted.
TH_EXCLAVES_SCHEDULER_REQUEST	0x4	Thread made an Exclaves scheduler request (e.g., wait, wake) from the xnu scheduler during RPC handling. Reentering Exclaves or returning to user space is not permitted.
TH_EXCLAVES_XNUPROXY	0x8	Thread is directly calling into xnuproxy. Reentering Exclaves or returning to user space is not permitted.
TH_EXCLAVES_SCHEDULER_CALL	0x10	Thread is directly calling into the Exclaves scheduler. Reentering Exclaves or returning to user space is not permitted.
TH_EXCLAVES_STOP_UPCALL_PENDING	0x20	Thread has called to stop upcall to XNU.

Table 7.3: Settable `thread_exclaves_state_flags_t` values in a thread. The flags indicate the current thread state regarding Exclave execution.

We will back-reference these found Exclaves thread states in our analysis of Exclaves communication mechanisms and try to use them to provide context regarding the different options for calling into Exclaves.

7.9.3 Calling into Exclaves – Mach Ports

The default mechanism for IPC are Mach ports. As found in Listing 7.1, every Exclave resource holds an `ipc_port_t` field `r_port`. We know IPC ports to be a named equivalent to Mach ports of type `mach_port_t` (`osfmk/ipc/ipc_port.h`, l. 150), which are the core building blocks of IPC in Apple operating systems. Mach ports serve as unidirectional communication endpoints for inter-task communication. They are associated with so-called `port rights` indicating allowed operations on the port (e.g., send or receive) for a specific task [4]. Whilst the exact workings of IPC via Mach can be deemed out of scope in this analysis, the existence of a Mach port on an Exclave resource indicates that it can be communicated with via standard IPC mechanisms.

We find the Mach port of Exclave resources to get set during the previously looked at `exclaves_resource_init` function via a call to `exclaves_resource_alloc` (`osfmk/kern/exclaves_resource.c`, l. 578). The function allocates a kernel object port of the Exclave-specific type `IKOT_EXCLAVES_RESOURCE` in the kernel IPC space and assigns it to the resource's port. The creation is shown in Listing 7.7.

```
1  /*
2      * Each resource has an associated kobject of type
3      * IKOT_EXCLAVES_RESOURCE.
4      */
5     ipc_port_t port = ipc_kobject_alloc_port((ipc_kobject_t)resource,
6         IKOT_EXCLAVES_RESOURCE, IPC_KOBJECT_ALLOC_NSREQUEST);
7     resource->r_port = port;
```

Listing 7.7: Kernel object port allocation for Exclaves resource in `exclaves_resource_alloc`.

This allocation does not yet register any send rights on the IPC port; therefore, no tasks can call towards it as of now. We find a function that actually registers a send right on an

Exclave resource port. This function is `exclaves_resource_create_port_name` in `osfmk/kern/exclaves_resource.c`. The function is shown in Listing 7.8.

```
kern_return_t
exclaves_resource_create_port_name(exclaves_resource_t *resource,
 ↪  ipc_space_t space,
    mach_port_name_t *name)
{
    assert3u(os_atomic_load(&resource->r_usecnt, relaxed), >, 0);

    ipc_port_t port = resource->r_port;

    ip_mq_lock(port);

    /* Create an armed send right. */
    kern_return_t ret = ipc_kobject_make_send_nsrequest_locked(port,
        resource, IKOT_EXCLAVES_RESOURCE);
    if (ret != KERN_SUCCESS &&
        ret != KERN_ALREADY_WAITING) {
        ip_mq_unlock(port);
        exclaves_resource_release(resource);
        return ret;
    }

    /*
     * If there was already a send right, then the port already has an
     * associated use count so drop this one.
     */
    if (port->ip_srights > 1) {
        assert3u(os_atomic_load(&resource->r_usecnt, relaxed), >, 1);
        exclaves_resource_release(resource);
    }

    ip_mq_unlock(port);

    *name = ipc_port_copyout_send(port, space);
    if (!MACH_PORT_VALID(*name)) {
        /*
         * ipc_port_copyout_send() releases the send right on failure
         * (possibly calling exclaves_resource_no_senders() in the
         * process).
         */
        return KERN_RESOURCE_SHORTAGE;
    }
    return KERN_SUCCESS;
}
```

Listing 7.8: The `exclaves_resource_create_port_name` in `exclaves_resource.c`.

The function is parameterized with a reference to a resource, an IPC space, and a reference to a `mach_port_name_t`. An IPC space is a tasks namespace for its associated ports and defines its IPC capabilities such as sending and receiving [48]. A Mach port name is "a local identity for a Mach port", defining rights held by a port in a specific

context provided by an implied namespace being task-specific (`osfmk/mach/port.h`). Highly simplified, but condensed to the key parts, the function registers a send right on the IPC port of the provided resource, and returns a name for it.

Looking for its invokers, we find it referenced from `_exclaves_control_trap`, the kernel mach trap handler for userspace Exclave functionality invocations. More precisely, the function is called in the handling of `EXCLAVES_CTL_OP_SENSOR_CREATE`, `EXCLAVES_CTL_OP_AUDIO_BUFFER_CREATE`, `EXCLAVES_CTL_OP_NAMED_BUFFER_CREATE`, and `EXCLAVES_CTL_OP_NOTIFICATION_RESOURCE_LOOKUP` commands. This indicates that for sensor, audio buffer, named buffer, and notification resources, communication via standard IPC is possible.

Without diving into the details of user space Exclave function invocation, a call to the mach trap with such a creation command does in its essence validate that the client may access the resource (by validating the resource is in the domain defined by the tasks Conclave) and then creates a send right on the specific resources Mach port, and returning a name to it in kernel IPC space. The process of send rights creation for a sensor resource Mach port is shown in Figure 7.6.

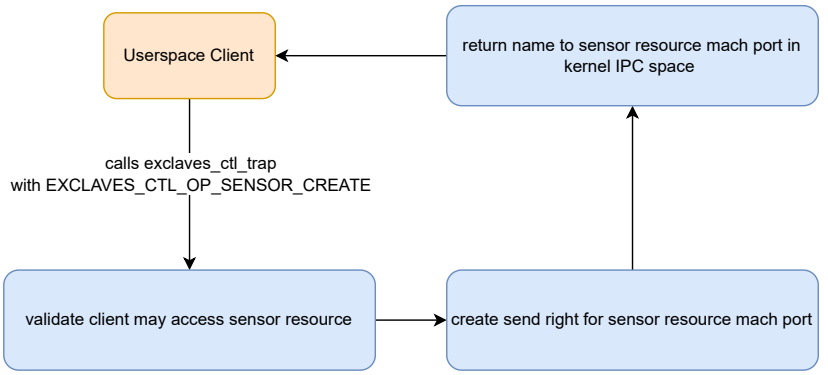


Figure 7.6: General control flow of `exclave_ctl_trap` call handling for sensor creation invocation.

It is important to note that this process does not provide the calling userspace client with a direct Mach port with send rights into the sensor resource, but instead provides him with the name associated with the port in the kernel IPC space. The kernel is now able to send messages to the sensor resource using this port, which is stored in the resources `r_port` field, and also associates the created port name with the sensor resource.

This is used when receiving userspace calls for sensor start, as such a call is parameterized with the previously returned name port. The kernel then verifies the caller's entitlement to start the sensor by attempting to retrieve a resource for the provided port name. It can be noticed that at no point in this shown communication the thread's `th_exclaves_state` field is changed. This confirms that the thread is not actually scheduled into Exclaves, but instead, communication relies on the standard IPC mechanism.

Even though calls into sensors, audio buffers, and named buffers appear to use standard IPC mechanisms, they are secured by the usage of Exclave domains. Only properly validated tasks, in terms of the associated Conclave domain, may access resources within that domain.

7.9.4 Calling into Exclaves – seL4-style IPC Buffer

We find that while specific Exclave resources (sensors, arbitrated buffers, audio buffers) use default IPC mechanisms, this does not hold for other resource types, such as services and the Conclave manager. We see no function invocations on these resources assigning their respective port with any rights. Therefore, a different communication interface must be used for communicating with Conclaves. For this, seL4 microkernel-style IPC buffers are employed.

seL4 is an open-source operating system microkernel within the L4 microkernel family. It is supported by the seL4 Foundation. It has a significant focus on security and is provably secure in terms of the security properties of confidentiality, integrity, and availability [31]. Although Apple's implementation differs, it shares core ideas and concepts.

The Exclaves IPC buffer definition can be seen in `osfmk/mach/exclaves_l4.h`:

```
/* ipc buffer object */
typedef struct __Exclaves_L4_Packed {
    /* message registers */
    Exclaves_L4_Word_t mr[Exclaves_L4_IpcBuffer_Mrs];
    /* source capability registers */
    Exclaves_L4_Word_t scr[Exclaves_L4_IpcBuffer_Crs];
    /* destination capability registers */
    Exclaves_L4_Word_t dcr[Exclaves_L4_IpcBuffer_Crs];
} Exclaves_L4_IpcBuffer_t;
```

Listing 7.9: Type definition for the Exclaves-specific IPC buffer.

In seL4, capabilities refer to "a unique, unforgeable token that gives the processor permission to access an entity or object in system" [46]. An instantiation of such a capability can be interpreted as a pointer to an object with specific access rights, and is in that sense comparable to a port right. We can therefore assume that the Exclaves IPC buffer can be used to transfer messages from XNU to Exclaves and back, as well as references to objects with access rights.

The assumption that XNU employs an IPC buffer for Exclaves communication is further supported by the fact that the `thread_t` type definition in `osfmk/kern/thread.h` lists a `*th_exclaves_ipc_buffer` field, which is the "Per-thread IPC buffer for Exclaves communication".

7.9.4.1 Exclaves IPC Buffer Allocation

Before proceeding further, we must perform a brief preemption regarding `xnuproxy`. As noted previously, `xnuproxy` is no longer a standalone binary in our analyzed firmware, but appears to have been integrated into the `sharedcache` binary. We will still refer to `xnuproxy` as a component, as we assume it still holds its central role regarding Exclaves management.

We know threads to be able to hold an `Exclaves_L4_IpcBuffer_t`. For a corresponding Exclaves IPC buffer threads do further hold a scheduling context id `th_exclaves_scheduling_context_id`, which is communicated to the exclave scheduling component (`osfmk/kern/thread.h`, l. 1007), and is required to correctly schedule requests (and with that, threads) into Exclaves via the IPC buffer. The process of a thread requesting and allocating an Exclaves IPC buffer is illustrated in Figure 7.7.

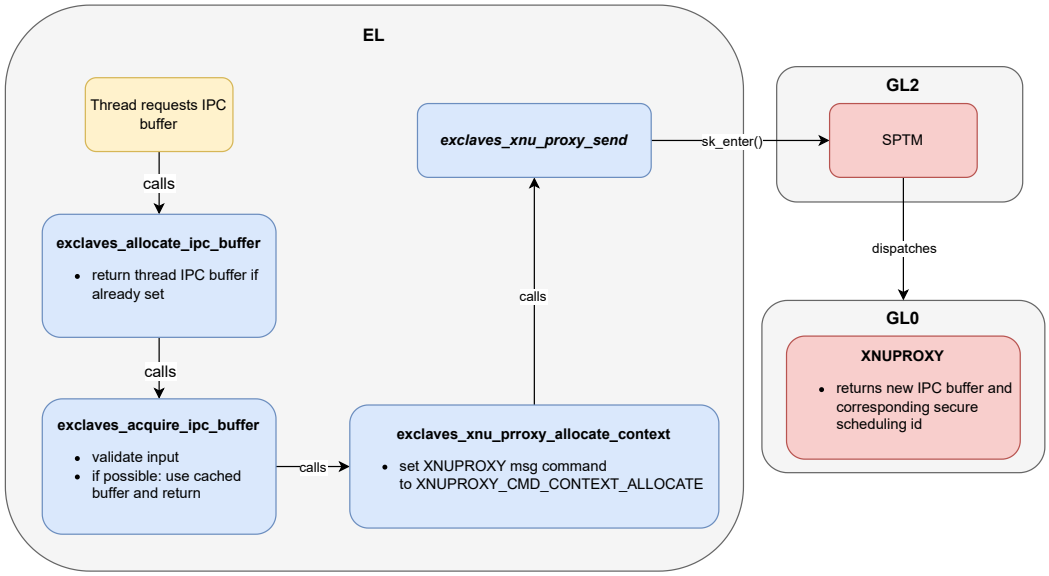


Figure 7.7: General control flow of an Exclave's IPC buffer request from a thread running in normal exception levels. The request is forwarded to `xnuproxy` in the secure world via an `sk_enter` call into SPTM.

The above figure shows the process of a thread requesting an IPC buffer. We will only consider the case for the initial IPC buffer request and ignore special cases, such as the reuse of cached buffers. Buffers are requested via an `xnuproxy` call with message command set to `XNUPROXY_CMD_CONTEXT_ALLOCATE`. This request is sent to SPTM via `sk_enter` and then dispatched to the Exclave GL0 management component, `xnuproxy`, which serves the request.

A more detailed look into `xnuproxy`, including how to call into it and request handling, will be provided in Section 7.9.5. As a result of the buffer request, the thread is provided with an IPC buffer and a corresponding scheduling context ID, which is known to `xnuproxy` and can be used to call into Exclaves.

Kernel threads request Exclaves IPC buffers on receiving an Exclaves endpoint call. This indicates that IPC buffers are the means of communicating endpoint calls to Exclaves.

7.9.4.2 Exclaves IPC Buffer Usage

Threads use Exclave IPC buffers to perform calls into Exclave endpoints. We can infer more detailed usage by examining the process of a user-space endpoint call to Exclaves. A shortened listing of the userspace Exclaves endpoint call wrapper is shown in Listing 7.10.

```
kern_return_t
exclaves_endpoint_call(mach_port_t port, exclaves_id_t endpoint_id,
    mach_vm_address_t msg_buffer, mach_vm_size_t size, exclaves_tag_t
     ↪ *tag,
    exclaves_error_t *error)
{
    kern_return_t kr = KERN_SUCCESS;
    if (size != Exclaves_L4_IpcBuffer_Size) {
        return KERN_INVALID_ARGUMENT;
    }
    Exclaves_L4_IpcBuffer_t *ipcb;
    ipcb = Exclaves_L4_IpcBuffer_Ptr((void*)msg_buffer);
    ipcb->mr[Exclaves_L4_Ipc_Mr_Tag] = *tag;
    const uint32_t opf = EXCLAVES_CTL_OP_AND_FLAGS(ENDPOINT_CALL, 0);
    kr = EXCLAVES_CTL_TRAP(port, opf, endpoint_id, msg_buffer, size, 0,
     ↪ 0);
    *tag = ipcb->mr[Exclaves_L4_Ipc_Mr_Tag];
    *error = EXCLAVES_XNU_PROXY_CR_RETVAL(ipcb);
    return kr;
}
```

Listing 7.10: Simplified Exclave endpoint call userspace wrapper in `libsyscall/wrappers/exclaves.c`.

To analyze such calls, we also need to consider the kernel-side handling, which is shown abbreviated for readability in Listing 7.11. The full listing can be found in Section A.18.

```
case EXCLAVES_CTL_OP_ENDPOINT_CALL: {
    Exclaves_L4_IpcBuffer_t *ipcb;
    if ((error = exclaves_allocate_ipc_buffer((void**)&ipcb))) {
        return error;
    }
    assert(ipcb != NULL);
    if ((error = copyin(ubuffer, ipcb, usize))) {
        return error;
    }

    /*
     * Verify that the service actually exists in the current
     * domain (only when the fallbacks are not enabled).
     */
    if (!exclaves_service_fallback &&
        !exclaves_conclave_has_service(task_get_conclave(task),
        identifier)) {
        return KERN_INVALID_ARGUMENT;
    }

    kr = exclaves_endpoint_call_internal(IPC_PORT_NULL, identifier);
    error = copyout(ipcb, ubuffer, usize);
   }
}
```

Listing 7.11: Simplified Exclave endpoint call handling in `osfmk/kern/exclaves.c`.

From `osfmk/kern/exclaves.h` we know the userspace wrapper `exclaves_endpoint_call` to be parameterized with a port (which has to be `IPC_PORT_NULL`), an endpoint ID being the identifier of the Exclave to send the RPC to, a tag parameter used "for exclaves IPC tag", and an error output parameter. We can infer the tag layout from its definition, as shown in Listing 7.12.

```
/* Exclaves_L4_MessageTag_t
 *
 *  32                                0
 *  llllllllllllllll...nuuucccrrrrrr
 *
 * r: message registers (6 bits)
 * c: capability registers (3 bits)
 * u: unwrapped capabilities (3 bits)
 * n: non-blocking (1 bit)
 * l: label (16 bits)
 */

typedef Exclaves_L4_Word_t Exclaves_L4_MessageTag_t;
```

Listing 7.12: `Exclaves_L4_MessageTag_t` definition in `osfmk/mach/exclaves_l4.h`.

Whilst the exact usage of the tag has not yet been fully reverse-engineered, we can assume the label of the tag works similarly to standard seL4 IPC message tags. We assume the kernel does not process it and instead passes it to the message target. It may hold information on a requested operation [47].

The `exclaves_endpoint_call` wrapper further receives a userspace buffer as an argument, for which it creates an Exclave's IPC buffer pointer, and sets the buffer pointer's tag message register to the user-provided tag. To do so, the user space buffer is interpreted as an `Exclaves_L4_IpcBuffer_Ptr`. This buffer is then provided to the kernel via the `_exclaves_ctl_trap` mach trap.

The kernel side handling of the request calls the previously analyzed `exclaves_allocate_ipc_buffer` to request a new Exclave IPC buffer. Then it copies the userspace-provided buffer contents, including the tag, into the newly allocated buffer. We assume that via this, userspace endpoint functionality requests are made available to exclaves.

After further verifying that the requested Exclave service exists, the actual call is performed via `exclaves_endpoint_call_internal`. This call encodes the requested Exclave resource ID into the IPC buffer and performs the actual call via a call to `exclaves_xnu_proxy_endpoint_call`. In this call, the thread Exclave state flag for `TH_EXCLAVES_RPC` is set, indicating that Exclave endpoint calls via IPC buffers are the Exclaves RPC mechanism. We will look at the actual `xnuproxy` call execution in Section 7.9.5. The general process of issuing an Exclaves endpoint call from user space and the kernel side handling until the `xnuproxy` call can be seen in Figure 7.8.

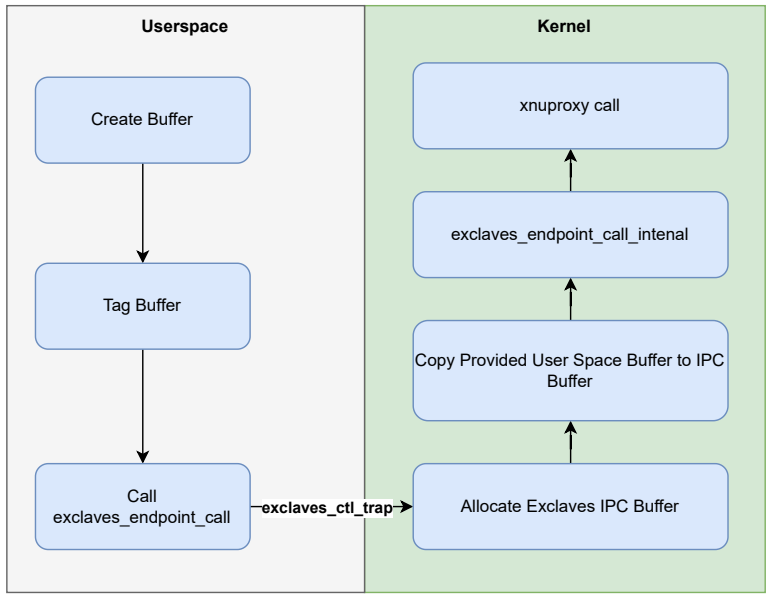


Figure 7.8: Process of issuing an Exclaves endpoint call from userspace and kernel side handling.

7.9.5 Xnuproxy Communication

The open source code parts of Exclaves show a multitude of references to `xnuproxy`. We initially assumed `xnuproxy` to be a specific GL0 component of the `Exclavecore`. We could confirm this in an older firmware version (22D82), where we found an individual `xnuproxy` binary. Interestingly, in the slightly newer firmware version that we focus on in this paper (22E40), this binary is no longer present. References to `xnuproxy` still

exist. Based on pattern and string matching, we find that `xnuproxy` was moved into the `sharedcache` binary of the `Exclavecore`. This further demonstrates that the Exclaves and entire secure world system are still in active development and subject to constant change. For the sake of better understandability, we will still refer to `xnuproxy` when discussing such components in the `sharedcache` binary.

From Table 7.3, we find `TH_EXCLAVES_XNUPROXY` to indicate a thread directly calling into xnuproxy. Except for directly calling into `xnuproxy`, we find it to be responsible for Exclave request forwarding. It receives both downcalls and upcalls and forwards them to the intended targets. Downcalls are calls from XNU into Exclaves, whilst upcalls denote calls back into XNU during Exclave execution, as a result of a required service offered by XNU. We can infer that `xnuproxy` is the respective handler for these from string references found in the `sharedcache` binary. An excerpt of the strings is shown in Listing 7.13.

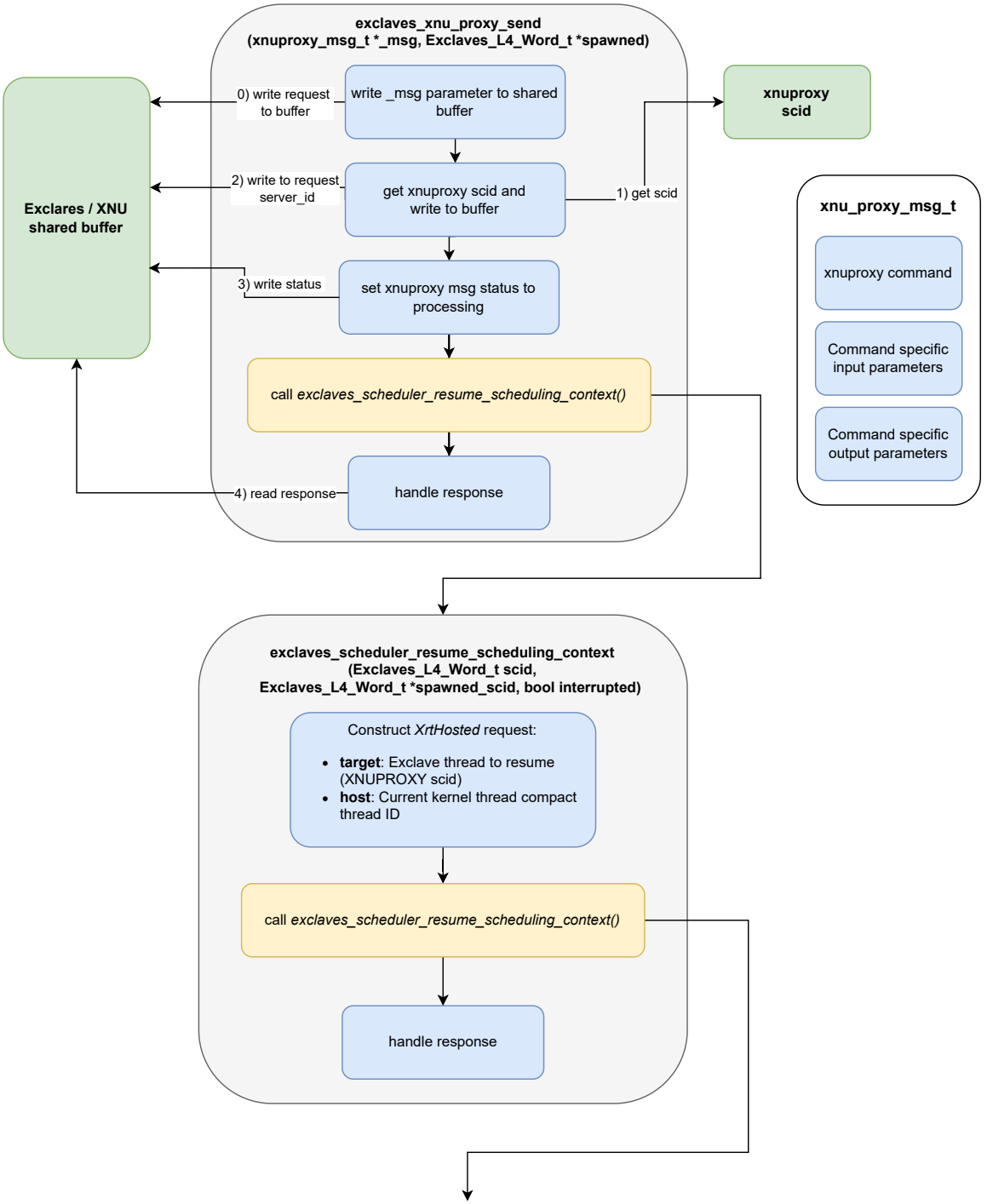
```
1  "[XNUP-D] Forwarding downcall for endpoint id %llu to cap 0x%lx with
    ↪  tag { words: %uh, capabilities: %uh, unwrapped: %uh, non_blocking:
    ↪  %s, label: 0x%04hx }\n"
2  ----------------------------------------------
3  "[XNUP-D] Forwarded downcall for endpoint id %llu returned %lu with tag
    ↪  { words: %uh, capabilities: %uh, unwrapped: %uh, non_blocking: %s,
    ↪  label: 0x%04hx }\n"
4  ----------------------------------------------
5  "[XNUP-U] Forwarding upcall to endpoint id %llu with tag
6  0x%lx badge %lx\n"
7  ----------------------------------------------
8  "[XNUP-U] Forwarded upcall to endpoint id %llu returned %lu with tag
    ↪  0x%lx\n"
9   ----------------------------------------------
10 "[XNUP-U] Invalid upcall endpoint id %llu, not forwarding and returning
    ↪  %lu\n"
```

Listing 7.13: Strings with `xnuproxy` reference from the sharedcache binary, allow for assumption with regards to its responsibilities. `[XNU-D]` appears to be downcall logging, and `[XNU-U]` upcall logging respectively.

`xnuproxy` calls are realized via the Exclaves IPC buffer, which allocations we looked at in Section 7.9.4.1.

7.9.5.1 Direct Calls into `xnuproxy`

Figure 7.9 shows the top-level control flow of a call from an XNU thread to `xnuproxy` directly as inferred from XNU open-source code. It serves as an overview of the more detailed process that we analyze in the following.


exclaves_xnu_proxy_send
(xnuproxy_msg_t *_msg, Exclaves_L4_Word_t *spawned)
write _msg parameter to shared buffer
0) write request to buffer
get xnuproxy scid and write to buffer
1) get scid
2) write to request server_id
set xnuproxy msg status to processing
3) write status
call exclaves_scheduler_resume_scheduling_context()
handle response
4) read response
Exclares / XNU shared buffer
xnuproxy scid
xnu_proxy_msg_t
xnuproxy command
Command specific input parameters
Command specific output parameters
exclaves_scheduler_resume_scheduling_context
(Exclaves_L4_Word_t scid,
Exclaves_L4_Word_t *spawned_scid, bool interrupted)
Construct XrtHosted request:
target: Exclave thread to resume (XNUPROXY scid)
host: Current kernel thread compact thread ID
call exclaves_scheduler_resume_scheduling_context()
handle response

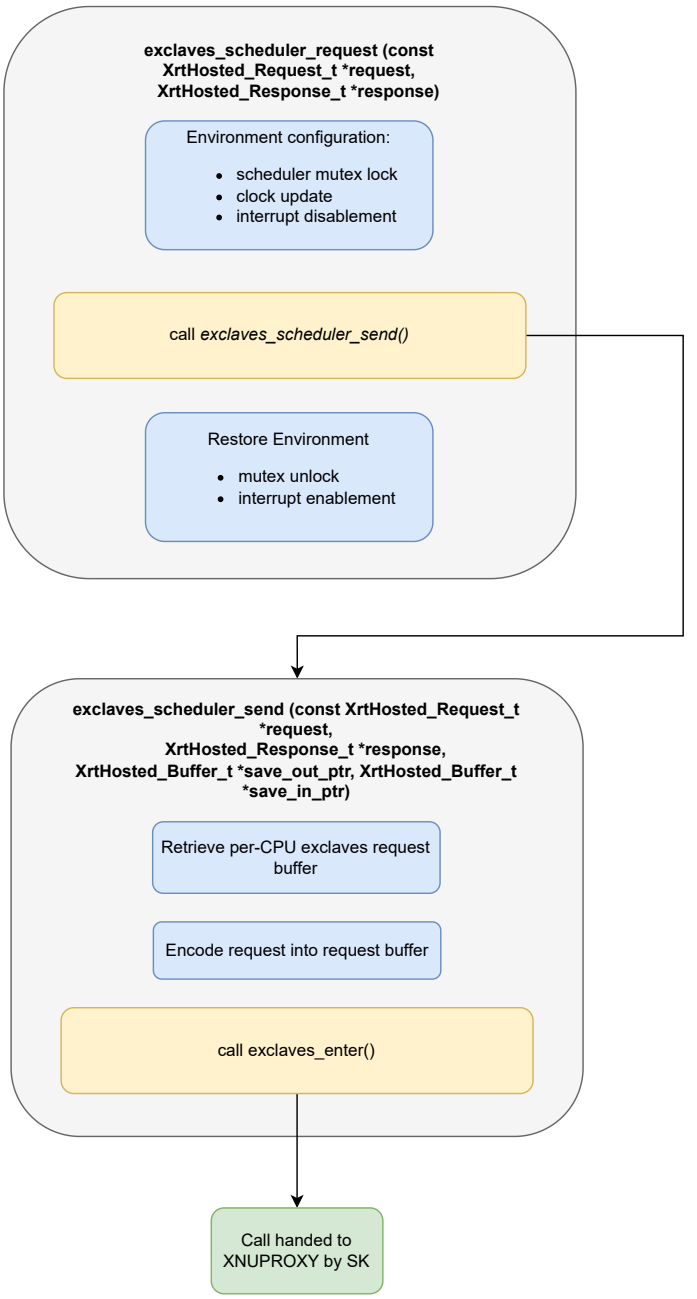


Figure 7.9: Control flow of a call to `xnuproxy` directly.

Direct calls to `xnuproxy` are found to be performed for management purposes. An example was seen earlier in Section 7.4. Such calls are parameterized with an `xnuproxy_msg_t` message, created with a command and what appears to be further command-specific input parameters. The creation of the message for the Exclave resource enumeration is shown in Listing 7.14.

```
xnuproxy_msg_t msg = {
            .cmd = xnuproxy_CMD_RESOURCE_INFO,
            .cmd_resource_info = (xnuproxy_cmd_resource_info_t) {
                .request.index = i,
            },
        };
```

Listing 7.14: Creation of `xnuproxy_msg_t` for Exclave resource enumeration in `exclaves_resource_init` (`osfmk/kern/exclaves_resources.c`).

We can enumerate all available `xnuproxy` management commands from a `cmd_to_str` function in `osfmk/kern/exclaves.c` (l. 2311). The full list can be found in Section A.19. We find a variety of commands similar to those callable from userspace (sensor create/start/stop, buffer copyout/create). Furthermore, there are commands that we have not yet seen before and assume to be relevant for Exclave management (buffer map, buffer layout, context allocate, context free).

exclaves_xnu_proxy_send The previously created `xnuproxy` message is passed as an input parameter to `exclaves_xnu_proxy_send` (`osfmk/kern/exclaves.c` l. 2402). The listing for this core function can be found in Section A.20. As the process of sending a message to `xnuproxy` appears to be quite complex, we will attempt to approach it from a high-level perspective.

To send a message directly to `xnuproxy`, the calling thread writes its request message to the shared Exclaves IPC buffer previously allocated (see Section 7.9.4.1). It further retrieves the globally stored `xnuproxy` Scheduling Context ID (SCID), which is set in the Exclaves setup process, and stores it in the Exclave buffers server ID field. This denotes the target of the call to perform. The call is progressed via `exclaves_scheduler_resume_scheduling_context`, which is parameterized with the `xnuproxy` SCID. Based on the input SCID and the calling threads Compact Thread ID (CTID), a so-called `XrtHosted_Response_t` is created, and input into the `exclaves_scheduler_request` call together with an output pointer. We find this function to be responsible for environment configuration in preparation for the call to xnuproxy, after which it forwards the call to `exclaves_scheduler_send`. This function finally writes the request to a per-CPU Exclaves request buffer structure and performs an `exclaves_enter` call, which ultimately leads to entering SPTM via a call to `sk_enter`.

Whilst we have not yet fully reverse-engineered the secure world request handling and thread scheduling, our previous analysis of SK provides us with a strong base for assumptions. From the `sk_enter` call, we know SK to get dispatched from SPTM. For an invoking `exclaves_enter` call, we know SK to hand control to a GL0 component (see Listing 6.4. We presume this component to be xnuproxy, which, upon invocation, recognizes that a request has been made and retrieves information via the shared buffer structures. The exact inner workings of this secure world side handling have not yet been reverse-engineered by us and are left as future work.

7.9.5.2 Exclaves Endpoint Calls

Next to explicitly calling into `xnuproxy` for management purposes, we find that `xnuproxy` is also responsible for performing endpoint calls to Exclaves. A kernel thread receiving an endpoint call via `exclaves_ctl_trap` serves this call by calling `exclaves_endpoint_call_internal` on the provided endpoint identifier to call,

which shows to be a wrapper for `exclaves_xnu_proxy_endpoint_call`. We find this call to store the provided endpoint ID in the thread's per-thread Exclaves IPC buffer endpoint field, and call `exclaves_scheduler_resume_scheduling_context` parameterized with the thread SCID. We assume this to be done to correctly schedule the thread once it reaches secure world. At this point, call handling is comparable to calls directly to xnuproxy. We assume that after receiving the call, `xnuproxy` identifies the call as a proper endpoint call via the set endpoint field in the IPC buffer, and performs forwarding for thread scheduling of the requested task into the secure world.

Xnuproxy Conclusions

We find `xnuproxy` to be responsible for calling into Exclaves running in the secure world. It appears to interplay with the Exclaves GLo `scheduler`, which is yet to be analyzed. It additionally serves as a GLo management component, being callable with various management commands. A key aspect of this is the provision of shared IPC buffer and scheduling context IDs to threads trying to call into exclaves, which appear to be required to perform such calls by scheduling the calling threads into the secure world. Furthermore, it acts in the capacity of a request forwarder in the secure world, dispatching incoming requests to the endpoints of the correct targets.

7.10 Tightbeam

Looking into Exclaves communication, we find a variety of references to Tightbeam. Tightbeam is an `ExclaveKit` private framework. It serves as an Exclave communication framework for secure world components. We find that XNU can also utilize it. The underlying transport mechanisms for Tightbeam are found to be `xnuproxy` for calls from XNU, which we previously found to perform Exclaves endpoint calls via Exclave dedicated IPC buffers.

7.10.1 Tightbeam Types

Before diving into a deeper analysis of Tightbeam's inner workings, it is essential to lay the fundamental groundwork by reverse-engineering Tightbeam's specific data types. We are aware of the existence of these types due to their usage in the XNU open-source code. We find the following Tightbeam types directly in the existing source:

- `tb_message_t`
- `tb_endpoint_t`
- `tb_client_connection_t`

We can partially reconstruct these via the Tightbeam binary by looking at getter and setter functions. Following this approach allows us to further define other Tightbeam-specific data types, which will be used in the reverse engineering of Tightbeam. This includes the following types:

- `tb_transport`
- `tb_connection`

The following will define these types as far as possible without official sources.

7.10.1.1 Tightbeam Message Type `tb_message_t`

We find multiple setter and getter functions for `tb_message_t` in the Tightbeam binary. The binary provides a large number of exported symbols, and these functions can be found by looking for the characteristic `_tb_message_set_<field>` and `_tb_message_get_<field>` methods. The reconstructed type definitions are shown in Table 7.4.

Offset	Field	Type	Notes
0x00	`state`	`uint32_t`	Message state value
0x04	`disposition`	`uint8_t`	Message disposition
0x08	`connection_identifier`	`uint64_t`	Connection ID
0x10	`client_identifier`	`uint64_t`	Client ID
0x18	`msg_identifier`	`uint64_t`	Message ID
0x48	`num_caps`	`uint64_t`	Unclear
0x50	`transport_buffer`	`tb_transport_message_buffer_t`	Transport buffer

Table 7.4: Inferred partial layout of `tb_message_t` based on reverse engineering of the Tightbeam framework.

We have further inferred on available values for the state and disposition field based on their usage in the Tightbeam binary and corresponding assertion failure strings. The detected values for the message state field are shown in Table 7.5.

Value	State
0	`TB_MESSAGE_STATE_UNINITIALIZED`
1	`TB_MESSAGE_STATE_PREPARING`
2	`TB_MESSAGE_STATE_READY`
3	`TB_MESSAGE_STATE_SENT`
4	`TB_MESSAGE_STATE_RECEIVED`

Table 7.5: `tb_message_t` state values detected in the Tightbeam binary.

The corresponding enumeration for the disposition field can be seen in Table 7.6.

Value	State
1	`TB_MESSAGE_DISPOSITION_QUERY`
2	`TB_MESSAGE_DISPOSITION_REPLY`

Table 7.6: `tb_message_t` disposition values detected in the Tightbeam binary.

7.10.1.2 Tightbeam Endpoint Type `tb_endpoint_t`

We can perform a similar type reconstruction based on Tightbeam binary getter and setter methods for the `tb_endpoint_t` type. The results can be seen below in Table 7.7.

7.10.1.3 Tightbeam Client Connection Type `tb_client_connection_t`

We find no direct setter and getter methods for the `tb_client_connection_t` type, and assume it to be a wrapper type around the more generic `tb_connection_t`, which will be looked at in Section 7.10.1.5

Offset	Field	Type	Notes
0x00	`type`	`uint32_t`	Endpoint type identifier
0x04	`options`	`uint32_t`	Endpoints options or flags
0x08	`interface_identifier`	`uint64_t`	Interface identifier
0x20	`data / value`	`uint64_t`	Payload data or value associated
0x28	`validity flag`	`uint8_t`	Inferred to by usage

Table 7.7: Inferred partial layout of `tb_endpoint_t` based on reverse engineering of the Tightbeam framework.

.

7.10.1.4 Tightbeam Transport Type `tb_transport_t`

We find no usage of this type in the XNU open-source code, but we can still infer its layout based on its few getter and setter methods. Despite large parts of the type being unknown, we still list it in Table 7.8 for completeness' sake.

Offset	Field	Type	Notes
0x08	`endpoint_data`	`uint64_t`	Data field retrieved from endpoint, inferred in Section 7.10.2.2
0x48	`vtable`	`uint64_t`	Pointer to vtable holding transport type specific function implementations, inferred in Section 7.10.2.2
0x50	`message_handler`	`uint64_t`	Message handler
0x68	`context`	`uint64_t`	Context

Table 7.8: Inferred partial layout of `tb_transport_t` based on reverse engineering of the Tightbeam framework.

7.10.1.5 Tightbeam Connection Type `tb_connection_type_t`

Similar to the `tb_transport_t` type before, we find some fields via setter and getter methods for the `tb_connection_type_t`. We include them anyway, and find a `transport` field in the datatype. We assume this to be of type `tb_transport_t`. The datatype can be seen below in Table 7.9.

Offset	Field	Type	Notes
0x0	`transport`	`uint64_t`	Transport, assumed type `tb_transport_t`
0x8	`observers`	`uint64_t`	Observers

Table 7.9: Inferred partial layout of `tb_connection_t` based on reverse engineering of the Tightbeam framework.

7.10.1.6 Tightbeam Transport Message Buffer Type `tb_transport_message_buffer_t`

A key type used by Tightbeam is `tb_transport_message_buffer_t`. While no fields were directly inferable from getter and setter methods, our reverse engineering revealed large parts of the structure.

Offset	Field	Type	Notes
0x0	`type`	`uint64_t`	Used to encode function to call at endpoint.
0x8	`wrapping`	`uint64_t`	
0x10	`offset`	`uint64_t`	Current offset into the buffer.
0x18	`size`	`uint64_t`	Total buffer size.

Table 7.10: Inferred partial layout of `tb_transport_message_buffer_t` based on reverse engineering of the Tightbeam framework.

The exact usage of the wrapping field remains unclear at this time. Our analysis will further demonstrate that the type field is used to encode a function identifier into the buffer.

7.10.2 Tightbeam Communication

We can reasonably assume that Tightbeam is a mechanism that supports calls towards Exclaves, and specifically Conclaves. This is supported by the fact that, as previously seen, every Conclave resource has a `tb_client_connection_t`, which we assume to be a Tightbeam communication interface (see Section 7.5). Tightbeam usage is largely redacted in the XNU open-source code. Still, we can infer its presence from the general structure of the available sections and create a more complete picture through direct reverse engineering.

We find various references to Tightbeam in `osfmk/kern/exclaves_driverkit.c` function `hello_driverkit_interrupts`, which appears to be a testing/debug build only test function. Whilst this does mean that the function will not be included in our production build firmware, it also offers more detailed insights into Tightbeam control flow. Communication via Tightbeam can be divided into several phases. An excerpt from the relevant function is shown below in Listing 7.15.

```
#define EXCLAVES_ID_HELLO_INTERRUPTS_EP              \
    (exclaves_service_lookup(EXCLAVES_DOMAIN_KERNEL, \
    "com.apple.service.HelloDriverInterrupts"))

    tb_endpoint_t ep = tb_endpoint_create_with_value(
        TB_TRANSPORT_TYPE_XNU, EXCLAVES_ID_HELLO_INTERRUPTS_EP, 0);

    tb_client_connection_t client =
        tb_client_connection_create_with_endpoint(ep);

    tb_client_connection_activate(client);
```

Listing 7.15: Simplified initial Tightbeam interaction in `hello_driverkit_interrupts` in `exclaves_driverkit.c`.

7.10.2.1 Endpoint Creation

The initial step of Tightbeam communication is the creation of an endpoint. In the above excerpt, this is realized via a call to `tb_endpoint_create_with_value`. The call is parameterized with a transport type,[13] an identifier, and an endpoint options parameter.[14]

The identifier appears to be an Exclave resource `r_id`, retrievable via a call to `exclave_service_lookup` with resource name and domain. The endpoint creation function has a return type of `tb_endpoint_t`. Its implementation is redacted from the XNU open-source code.

We find the function with a symbolicated name implemented in the Tightbeam binary. The reassembled function is shown in Listing 7.16.

```
void _tb_endpoint_create_with_value(undefined4 type,long id,undefined4
 ↪  options)
{
  tb_endpoint_t *puVar1;

  puVar1 = (tb_endpoint_t *)0x1;
  func_0x00026258b090(1,0x60,0x1082040faca7f44);
  if (puVar1 != (tb_endpoint_t *)0x0) {
    puVar1->type = type;
    puVar1->options = options;
    puVar1->data = id;
    puVar1->validity_flag = 1;
    return;
  }
  TB_ASSERT_PTR_!=_NULL_FAIL();
}
```

Listing 7.16: The `_tb_endpoint_create_with_value` function in Tightbeam, symbolicated by hand. The source code was disassembled by Ghidra.

The function appears to show a memory allocation via a function call to a non-resolved function. This function is not resolved in the binary. After the successful allocation, an endpoint of type `tb_endpoint_t` is initialized with the provided parameters. Although not clearly visible in the reassembled code excerpt, based on usage, we assume the call returns the created endpoint.

Based on the implementation in Tightbeam, its control flow, and assumed usage in the kernel, we can find the respective function implementation in the kernel. The reassembled code excerpt can be seen in Listing 7.17. We reconstructed symbols, in particular types, variable names, and functions.

[13] Transport type is `TB_TRANSPORT_TYPE_XNU` for all XNU invocations.
[14] Transport options are `TB_ENDPOINT_OPTIONS_NONE` or 0 for all XNU invocations.

```
tb_endpoint_t * tb_endpoint_create_with_value(undefined4 type,long
 ↪  id,undefined8 ep_options)

{
  tb_endpoint_t *endpoint;
  undefined4 *puVar1;
  undefined8 ep_options_copy;
  undefined4 in_w3;
  undefined1 auVar2 [12];

  ep_options_copy = ep_options;
  endpoint = (tb_endpoint_t
   ↪  *)allocate_in_structure(&struct_tb_endpoint_s,4);
  if (endpoint != (tb_endpoint_t *)0x0) {
    endpoint->type = type;
    endpoint->options = (int)ep_options;
    endpoint->data = id;
    endpoint->validity_flag = 1;
    return endpoint;
  }
  auVar2 = TB_ASSERT_PTR_!=_NULL_FAIL();
  puVar1 = auVar2._0_8_;
  *puVar1 = auVar2._8_4_;
  puVar1[1] = in_w3;
  *(undefined1 *)(puVar1 + 10) = 0;
  *(undefined8 *)(puVar1 + 8) = ep_options_copy;
  return (tb_endpoint_t *)0x0;
}
```

Listing 7.17: `tb_endpoint_create_with_value` in the kernel binary, symbolicated by hand. The source code was disassembled by Ghidra.

The previously redacted allocation function in the Tightbeam binary is shown to be an allocator to a passed structure in the kernel implementation. Otherwise, the kernel function appears to be similar to the one found in the Tightbeam binary. Having found the relevant function implementation in the kernel, we can infer that the integer value for `TB_TRANSPORT_TYPE_XNU` is 7. This is based on all detectable calls to the function in the kernel. It should be noted that the input identifier, a unique Exclave resource identifier, is written to the endpoint data field. A resulting endpoint structure is shown in Figure 7.10.

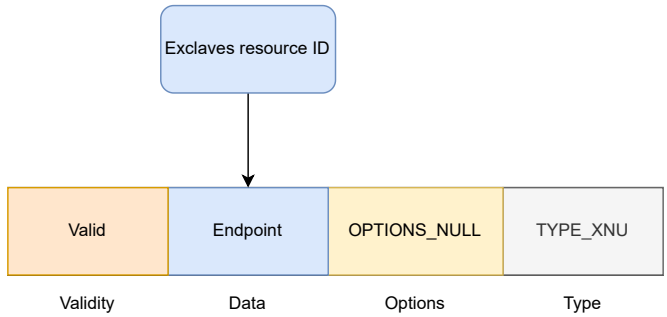


Figure 7.10: Tightbeam `tb_endpoint_t` endpoint created after call to `tb_endpoint_create_with_value` in XNU.

7.10.2.2 Client Connection Creation

Recalling Listing 7.15, the newly created endpoint is now used as an input parameter in a function call to `tb_client_connection_create_with_endpoint`. We find the relevant function in XNU and can label it, as well as its called functions, based on the implementation in Tightbeam. The function implementation in the kernel can be seen in Listing 7.18.

```
tb_connection_t *
 ↪  tb_client_connection_create_with_endpoint(tb_endpoint_t *ep)
{
  tb_connection_t *ptVar1;
  __tb_connection_create_transport_for_endpoint(ep,0);
  ptVar1 = (tb_connection_t *)xnu_tb_connection_create();
  xnu_tb_endpoint_destruct(ep);
  return ptVar1;
}
```

Listing 7.18: `tb_client_connection_create_with_endpoint` in the kernel binary, symbolicated by hand. The source code was disassembled by Ghidra.

The above listing has been symbolized in the kernel binary based on the function implementation in Tightbeam. The `__tb_connection_create_transport_for_endpoint` function creates a `tb_transport_t` value based on the input endpoint. In both the kernel and Tightbeam implementations, the function implements a switch on the input endpoint transportation type. We can recall that this was an integer value of 7 for `TB_TRANSPORT_TYPE_XNU`. Interestingly, while the kernel implementation supports and implements integer endpoint type value 7, the Tightbeam implementation does not. This indicates that different types of Tightbeam transports can only be created from specific contexts. The reassembled function implementation from Tightbeam can be found in Section A.21. It allows us to infer various types of Tightbeam transport.

It shows that Tightbeam supports a variety of different transport types. As already noted, `TB_TRANSPORT_TYPE_XNU` is not implemented in the Tightbeam binary. It is implemented in the kernel in `__tb_connection_create_transport_for_endpoint`. The reassembled implementation is shown in Listing 7.19.

Type	Inferred Transport Type Name	Flag	Creation Function
1	TB_TRANSPORT_TYPE_NULL	any	_tb_null_transport_create
2	TB_TRANSPORT_TYPE_MACH	1	_tb_mach_service_transport_create
2	TB_TRANSPORT_TYPE_MACH	0	_tb_mach_client_transport_create
4	TB_TRANSPORT_TYPE_EVE	0	_tb_eve_client_transport_create
5	TB_TRANSPORT_TYPE_EVE	0	_tb_eve_client_transport_create
8	TB_TRANSPORT_TYPE_DARWIN	any	_tb_darwin_client_transport_create
9	TB_TRANSPORT_TYPE_UNIX	0	_tb_unix_client_transport_create_with_endpoint
9	TB_TRANSPORT_TYPE_UNIX	1	_tb_unix_service_transport_create_with_endpoint
10	TB_TRANSPORT_TYPE_DELEGATED	0	_tb_delegated_client_transport_create
10	TB_TRANSPORT_TYPE_DELEGATED	1	_tb_delegated_service_transport_create
11	TB_TRANSPORT_TYPE_AFK	any	_tb_afk_transport_create

Table 7.11: Mapping from endpoint type and endpoint options to inferred transport type name and the relevant transport creation function from `__tb_connection_create_transport_for_endpoint` in Tightbeam.

```
void __tb_connection_create_transport_for_endpoint(tb_endpoint_t *ep)

{
  code *pcVar1;
  int iVar2;
  long lVar3;
  undefined8 extraout_x1;
  undefined8 in_x2;
  undefined4 in_w3;

  iVar2 = tb_ep_get_type();
  if (iVar2 == TB_TRANSPORT_TYPE_AFK) {
    lVar3 = _tb_afk_transport_create(ep);
  }
  else {
    if (iVar2 != TB_TRANSPORT_TYPE_XNU) goto LAB_ffffff0088ed670;
    lVar3 = tb_xnu_transport_create(ep,extraout_x1,in_x2,in_w3);
  }
```

Listing 7.19: Simplified `__tb_connection_create_transport_for_endpoint` disassembly of the kernel binary. The source code was disassembled by Ghidra.

XNU appears to only support `TB_TRANSPORT_TYPE_XNU` and `TB_TRANSPORT_TYPE_AFK`. The transport creation function for the XNU type has been named according to the naming convention used in the Tightbeam binary. `tb_xnu_transport_create` performs a two-step allocation process. The resulting structure of this allocation process can be seen in Figure 7.11

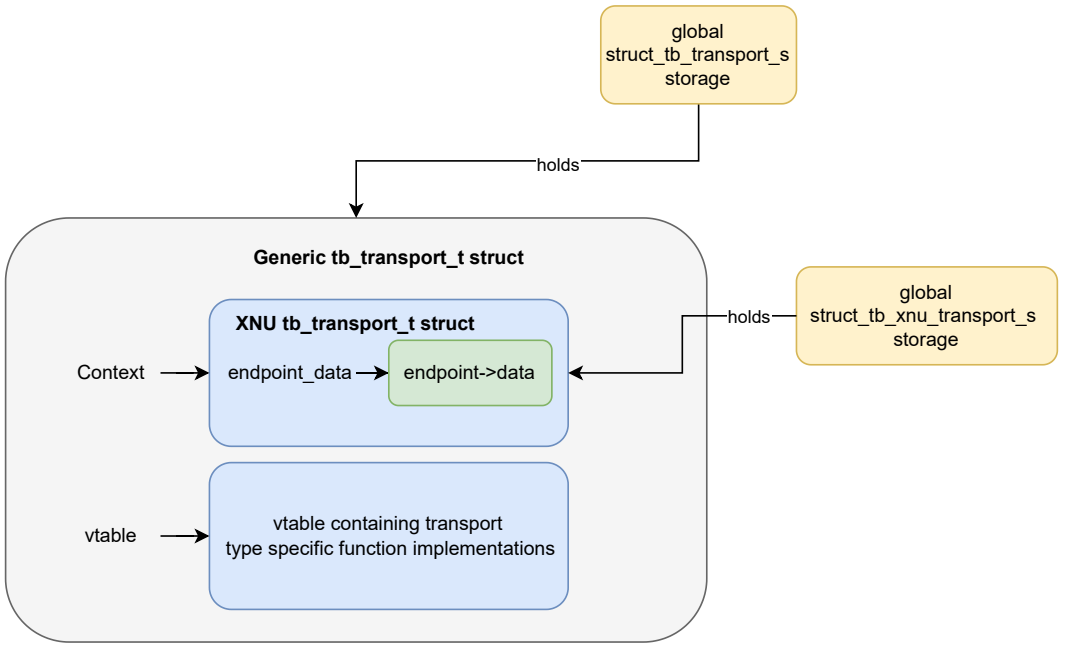


Figure 7.11: Resulting structure of `tb_xnu_transport_create` in XNU.

Initially, a generic `tb_transport_t` struct is allocated in a global transport storage. A second XNU type-specific struct is allocated and stored in a further global XNU transport storage. The xnu transport is written to the context field of the generic struct. Furthermore, the data value (recall that this corresponds to the Exclave's resource ID) is written to a field in the XNU struct, which we have denoted `endpoint_data`. Furthermore, a vtable pointer is written to a field of the generic struct, which we have denoted `vtable`. We know this to be a vtable based on a similar implementation in the Tightbeam binary, in which the tables are symbolicated and named. The XNU vtable holds a variety of different function pointers. Confirming to the standard, it appears that vtables are used to implement transport type polymorphisms by storing the type-specific functions in the generic transport structure. The XNU Tightbeam transport vtable can be seen below in Listing 7.20.

```
tb_xnu_client_transport_vtable
        tb_xnu_transport_send_message
        LAB_fffffff0088f1090
        __tb_connection_create_transport_for_endpoint
        FUN_fffffff0088f10e0
        __tb_xnu_transport_destruct_message_buffer
        LAB_fffffff0088f1194
        FUN_fffffff0088f11bc
        FUN_fffffff0088f11cc
        FUN_fffffff0088f11d8
```

Listing 7.20: Simplified function entries in XNU transport type-specific vtable stored in the generic transport struct in the kernel. Partially symbolized by hand. The source code was disassembled by Ghidra.

After the transport has been successfully created, a `tb_connection_t` is created via a call to `xnu_tb_connection_create`. The function allocates a connection in a global `struct_tb_connection_s` storage, and stores the previously created transport, and a newly allocated `tb_list_t`. The exact usage of the list is still unknown.

To finish up the handling of `tb_client_connection_create_with_endpoint`, the previously created endpoint is destroyed, as it has served its purpose and is no longer needed. A `tb_client_connection_t` is finally returned to the caller. The result of this part of the Tightbeam control flow is shown in Figure 7.12.

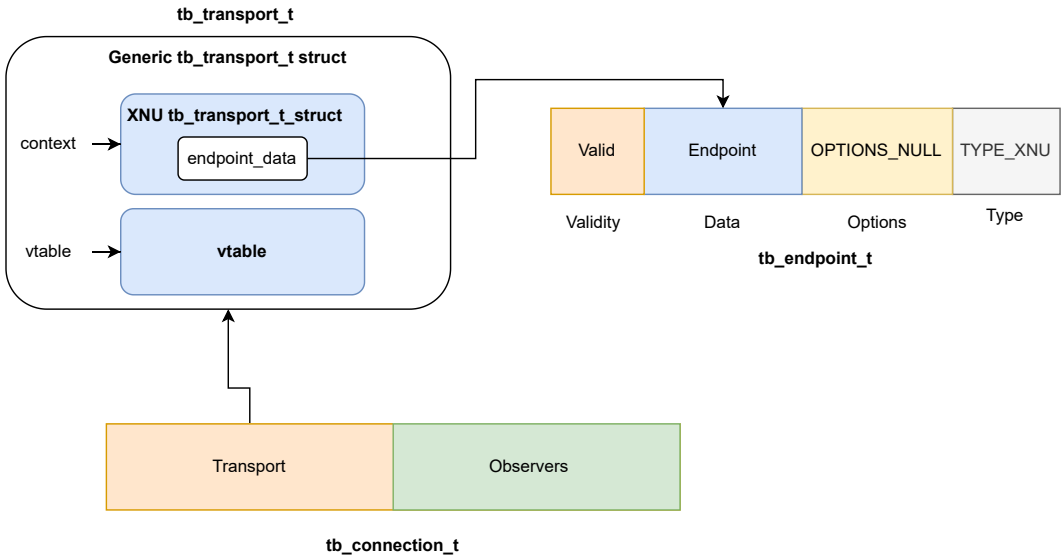


Figure 7.12: Resulting structure of `tb_client_connection_create_with_endpoint` call from XNU. A `tb_connection_t` is returned to the caller.

7.10.2.3 Client Connection Activation

Recalling Listing 7.15, we see that the next step in the Tightbeam communication process is the activation of the previously created connection via a call to `tb_client_connection_activate`. The function can be seen in Listing 7.21.

```
void tb_client_connection_activate(tb_connection_t *client_connection)
{
  long lVar1;
  tb_transport_t *transport;

  transport = (tb_transport_t *)client_connection->transport;
  if ((client_connection->observers != 0) &&
     (lVar1 = *(long *)(client_connection->observers + 8), lVar1 != 0))
      ↪ {
    (**(code **)(lVar1 + 0x10))(lVar1 + 0x10,lVar1,transport);
  }
  activate_connection(transport);
  return;
}
```

Listing 7.21: `tb_client_connection_activate` implementation in the kernel binary, symbolicated by hand. The source code was disassembled by Ghidra.

Conditional code execution is based on the existence of observers. We have not yet reverse-engineered this part of the Tightbeam framework, but assume this to be an informing call to registered observers that the connection is now active. The actual activation of the connection happens via a function call to what we have denoted `activate_connection`, which receives the connection transport as an input. It can be seen in Listing 7.22.

```
void tb_client_connection_activate(tb_connection_t *client_connection)
{
bool activate_connection(tb_transport_t *param_1)

{
  int iVar1;

  if ((tb_transport_t *)param_1->vtable != (tb_transport_t *)0x0) {
    vtable = (tb_transport_t *)param_1->vtable;
  }
  iVar1 = (*(code *)vtable->field_0x8)();
  return iVar1 != 0;
}
}
```

Listing 7.22: `connection_activate` implementation in the kernel binary, symbolicated by hand. The function code has been slightly altered for readability. The source code was disassembled by Ghidra.

The function accesses the transport vtable and executes the function registered at offset 0x8, which we assume to be the relevant activation function. Interestingly, we find that the specific function is implemented as an empty stub, which returns without taking any action. We assume this to indicate that connections for the XNU transport type do not require activation. In contrast, the specific vtable function in other transport-type vtables, found within the Tightbeam binary, at least partially implements actual functionality.

7.10.2.4 Message Preparation

After the Tightbeam setup for the client connection has been completed, the actual message delivery can be prepared and performed. The relevant code excerpt is listed in Listing 7.23.

```
tb_message_t message = NULL;
tb_transport_message_buffer_t tpt_buf = NULL;

message = kalloc_type(struct tb_message_s, Z_WAITOK | Z_ZERO |
 ↪  Z_NOFAIL);
tpt_buf = kalloc_type(struct tb_transport_message_buffer_s,
    Z_WAITOK | Z_ZERO | Z_NOFAIL);

tb_error_t tb_err = TB_ERROR_SUCCESS;
tb_err = tb_client_connection_message_construct(client, message,
    tpt_buf, sizeof(uint8_t), 0);

tb_message_encode_u8(message, (uint8_t) test_type);

tb_message_complete(message);

tb_message_t response = NULL;

tb_err = tb_connection_send_query(client, message, &response,
    TB_CONNECTION_WAIT_FOR_REPLY);
```

Listing 7.23: Message preparation and call delivery in `hello_driverkit_interrupts`.

The initial step is the allocation of memory for both the actual message and the `tpt_buf`, which we assume to be the transport buffer. After this, the message is constructed via a call to `tb_client_connection_message_construct`. This is a wrapper function for `__tb_connection_message_construct`, which is listed in Listing 7.24.

```
undefined8
__tb_connection_message_construct
          (tb_connection_t *connection,int option,tb_message
           ↪  *message,tb_bufffer *tpt_buffer,
          ulong size,undefined8 flag)
{
  int iVar1;
  ulong uVar2;
  undefined8 uVar3;
  undefined1 disposition;
  tb_transport_t *transport;

  tb_message_initialize(message);
  transport = (tb_transport_t *)connection->transport;
  iVar1 = tb_transport_supports_multipart_messages(transport);
  if (iVar1 == 0) {
    if (tpt_buffer->wrapping != '\0') goto
     ↪  TPT_BUFFER_WRAPPING_NOT_FALSE;
  }
  else {
    uVar2 = tb_transport_get_tx_buffer_size(transport);
    if (tpt_buffer->wrapping != '\0') {
TPT_BUFFER_WRAPPING_NOT_FALSE:
      __tb_connection_message_construct.cold.1();
      __tb_connection_message_destruct();
      return 0;
    }
    if (uVar2 < size) {
      _tb_connection_alloc_init_owned_transport_message_buffer(tpt_buff
       ↪  er,size);
      goto LAB_fffffff0088ee1e8;
    }
  }
  uVar3 = tb_transport_construct_message_buffer(transport,size,flag,tpt
   ↪  _buffer);
  if ((int)uVar3 != 0) {
    return uVar3;
  }
LAB_fffffff0088ee1e8:
  disposition = TB_MESSAGE_DISPOSITION_QUERY;
  if (option == 1) {
    disposition = TB_MESSAGE_DISPOSITION_REPLY;
  }
  uVar3 = _tb_message_construct(message,tpt_buffer,disposition);
  if ((int)uVar3 == 0) {
    tb_message_set_connection_identifier(message,connection);
    uVar3 = 0;
  }
  return uVar3;
}
```

Listing 7.24: Message construction via `__tb_connection_message_construct` in the kernel binary.

The function initializes an empty Tightbeam message. It further constructs a message buffer for the connection's transport by calling `tb_transport_construct_message_buffer`. This invokes a function at offset `0x18` in the transport vtable. This function initializes the buffer and sets, among others, the size and wrapping field. Based on the provided options, the message disposition is set, differentiating between a query and a reply. Finally, the message is constructed, and the constructed transport buffer and connection identifier fields are set. The message now holds the transport buffer, a disposition determining its communication state, and an identifier for the connection to which it belongs.

Message Encoding The `_tb_message_encode_<Encoding>` function in Tightbeam performs the actual data storage to the message transport buffer. By doing so, they make data available to the underlying call target. Based on invocations in the XNU open-source code, we see calls encoding function specifiers (in `exclaves_driverkit.c`) or raw byte data derived from a string (in `exclaves_test.c`). We can assume it can store arbitrary data, as long as it fits within the buffer's constraints.

Message Finalization The message is further finalized via a call to `tb_msg_complete`. This function sets the message state depending on the current state and the message disposition. For a query message, the state is set to `TB_MESSAGE_STATE_READY`, indicating its readiness to be sent.

7.10.2.5 Message Delivery

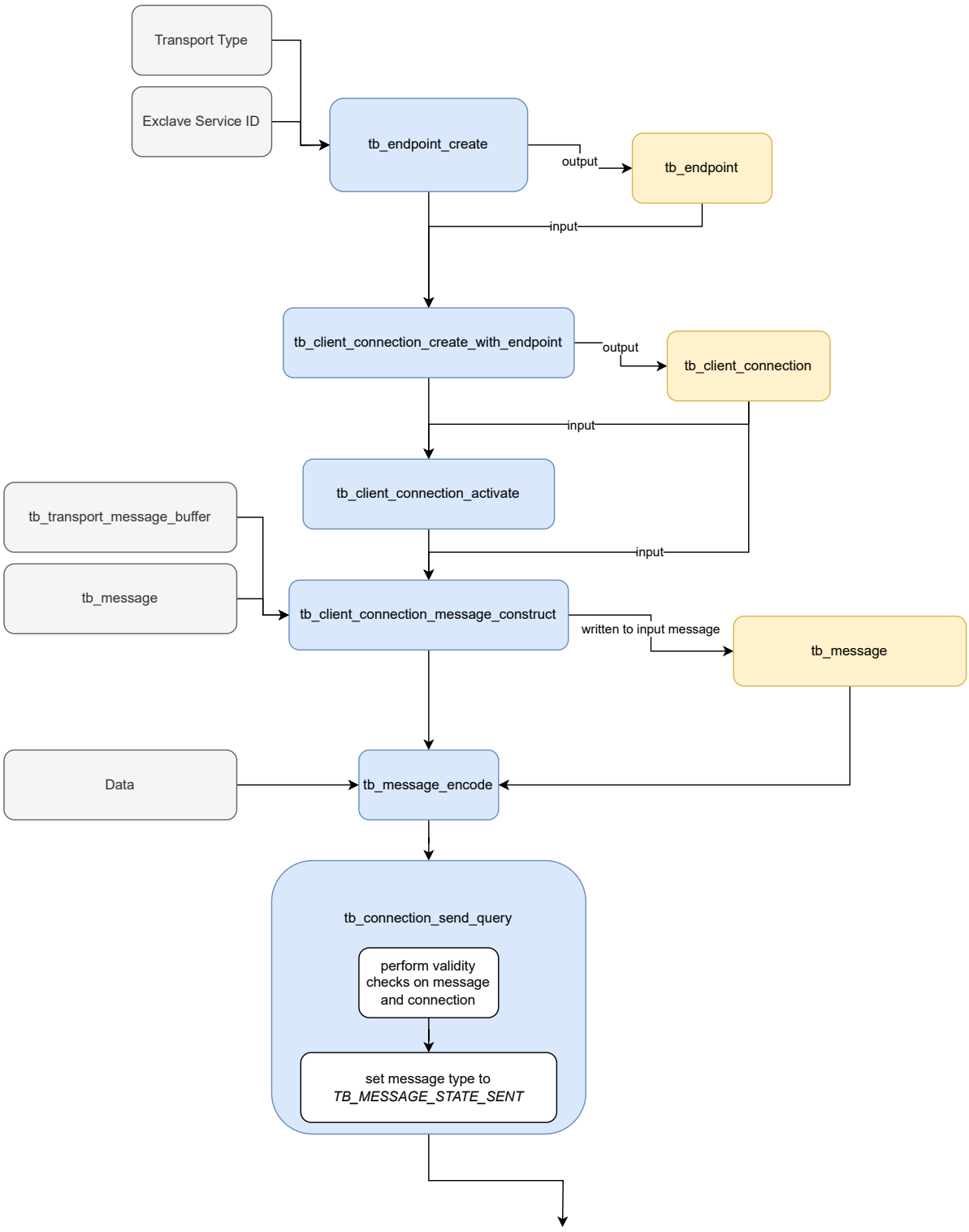
Finally, the actual message delivery is performed via a call to `tb_connection_send_query`. The full function Listing can be found in Listing A.14. The function performs sending validation by confirming the message state, disposition, and connection identifier. There is further setup occurring that has not yet been fully reverse-engineered; however, we know the actual message sending to be invoked by a function call to `tb_transport_send_message`, which in turn executes the first function entry in the transport vtable. For XNU, this function is `tb_xnu_transport_send_message`. An excerpt from that function is shown in Listing 7.25.

```
undefined8
tb_xnu_transport_send_message
          (tb_transport_t *transport,tb_message *message,undefined8
           ↪  *param_3,uint param_4)

{
  ulong uVar1;
  int iVar2;
  tb_bufffer *message_buffer;
  long lVar3;
  long error;
  ulong tag;

  if (message->msg_state != TB_MESSAGE_STATE_SENT) {
    FUN_ffffff0088f1418();
    return 0;
  }
  message_buffer = (tb_bufffer
   ↪  *)tb_message_get_transport_buffer(message);
  tag =
       (ulong)((int)message_buffer->size + 7U >> 3) & 0x3f |
       (ulong)*(ushort *)&message_buffer->someFlags << 0x10;
  output = 0;
  iVar2 = exclaves_endpoint_call
                    (*(undefined8 *)transport->context,transport->conte
                     ↪  xt->endpoint_data,
                     &tag,&error);
```

Listing 7.25: Excerpt from `tb_xnu_transport_message` in the kernel binary. The source code was disassembled by Ghidra.

We can determine that the called function is `exclaves_endpoint_call`, the kernel's Exclaves entry point, based on its calling context. As known from the open source code, it performs a call to Exclaves via `exclaves_xnu_proxy_endpoint_call`.

We can infer from this that `xnuproxy` is the underlying communication mechanism for Tightbeam communication from XNU to Exclaves. We furthermore find that the call from `tb_xnu_transport_send_message` is the only invocation of the kernel Exclaves entry point. We can therefore reasonably assert that Tightbeam is in fact the transport interface used by XNU to call into Exclaves. We have provided Figure 7.13 as a full overview of the process of sending a message to an Exclave endpoint via Tightbeam from XNU.


Transport Type
Exclave Service ID
tb_endpoint_create
output
tb_endpoint
input
tb_client_connection_create_with_endpoint
output
tb_client_connection
input
tb_client_connection_activate
tb_transport_message_buffer
tb_message
input
tb_client_connection_message_construct
written to input message
tb_message
Data
tb_message_encode
tb_connection_send_query
perform validity checks on message and connection
set message type to TB_MESSAGE_STATE_SENT

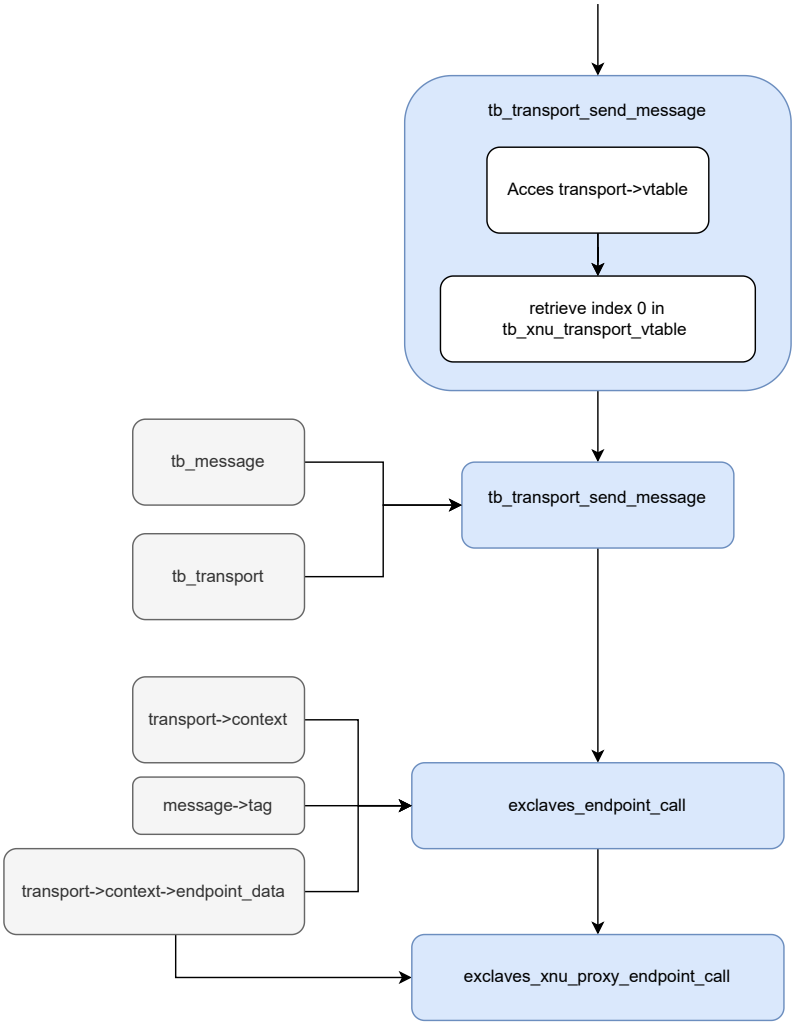


Figure 7.13: Control flow of setup of Tightbeam connection and performing a call to a specified endpoint.

7.10.3 Tightbeam Usage & Context

The reverse-engineered Tightbeam calling process was inferred from the structure provided in XNU. We do not find any direct usage of Tightbeam communication in the open-source code. From Section 7.5 we know that at Conclave initialization, each Conclave gets set up with a Tightbeam connection. Additionally, we find a large number of calls to our previously discovered `tb_connection_send_query` method call in the kernel binary, which indicates that the kernel does actively use Tightbeam for message delivery. Tracing back its calling functions, we find that a variety of these uses originate from the aforementioned Conclave setup process. Further invocations appear on `exclave_ctl_trap` Exclaves boot handling and invocations regarding operations on audio buffers and memory. Tightbeam is the communication framework used for performing Exclaves calls from the kernel directly, and therefore used whenever Exclaves

services are to be invoked. We find that the actual delivery via the underlying Exclaves IPC buffers is performed by `xnuproxy`.

Thightbeam is an Exclave communication framework, which wraps the underlying transport mechanism from the caller. References to Tightbeam are found in various GL0 Exclavecore components, which support our assumption. We have performed an in-depth analysis of Tightbeam calls executed from kernel; however, recalling Section 7.10.2.2, we found a variety of different transport types supported by Tightbeam. Tightbeam appears to act as an IPC mechanism with relatively rigid structuring regarding the underlying transport type. This might make it suitable for communication across security boundaries.

7.11 Real-World Exclave Use Case – Secure Indicator Light / Privacy Indicator Manipulation

With the iOS 14 release in September 2020, Apple introduced a so-called *recording indicator*. This indicator is displayed "whenever an app has access to the microphone or camera", aiming to provide increased security against spyware by rendering a highly visible dot on the device screen [49]. With the release of Exclaves, the actual implementation of the indicator light has shifted significantly. In an attempt to highlight security improvements through the implementation and usage of Exclaves, we will examine the initial implementation of the security indicator and explore ways to circumvent it. We compare this with the new implementation using the Exclaves feature, and aim to derive security implications from this.

7.11.1 Recording Indicator

The *recording indicator* refers to the initial version of the indicator light released in iOS 14. In the following, we will briefly examine its implementation and then demonstrate a way to circumvent it on devices without Exclaves.

7.11.1.1 Implementation

We assume the recording indicator to be realized by the `mediaserverd`, the daemon responsible for media streaming and sensor access in older iOS versions.

Although the official indicator name after the release was *recording indicator*, a `frida-trace` call for this was unsuccessful in `mediaserverd`. However, especially in the macOS context, the name *privacy indicator* circles in official Apple support material.[15] Tracing for Objective-C method names containing `PrivacyIndicator` displays `isEligibleForPrivacyIndicator` in `STMediaStatusDomainCameraDescriptor` and `initWithCameraIdentifier::eligibleForPrivacyIndicator` in `STMediaStatusDomainCameraDescriptor`. Unfortunately, overwriting these methods does not lead to a successful deactivation of the indicator, suggesting a more internal embedding and a need to check for its necessity in rendering.

Digging deeper and following the hints provided by the initial trace, tracing for `*[STMedia* *]` points towards the `STMutableMediaStatusDomainData`, indicating an alterable parameter set. Tracing for methods of said class leads to `-[STMutableMediaStatusDomainData initWithAttributionCatalog:cameraAttributionListData:]`, which one can assume initializes camera usage with a mutable set of parameters. The following section will show that this is, in fact, the case, and this data set is at least partially responsible for the rendering of the recording indicator.

[15] See https://support.apple.com/en-gb/guide/mac-help/mchlp1446/14.0/mac/14.0, accessed: 15.05.2025.

7.11.1.2 Circumvention

As seen in the previous subsection, the recording indicator is realized in the mediaserverd. This is a daemon running in the iOS userspace. We can therefore inject directly into it via FRIDA [30]. The following proof of concept was performed on an iPhone 8 running iOS 16.7.10. To inject via FRIDA, the device was jailbroken using the palera1n jailbreak [50] in rootless mode. The following FRIDA script was injected into the `mediaserverd` process.

```
// Usage: frida-trace -U mediaserverd -m
 ↪  '*[STMutableMediaStatusDomainData
 ↪  initWithAttributionCatalog:cameraAttributionListData:]'
// Script Location: Default handler location for
 ↪  STMutableMediaStatusDomainData
 ↪  initWithAttributionCatalog:cameraAttributionListData:
defineHandler({
  onEnter(log, args, state) {
    log(`-[STMutableMediaStatusDomainData
     ↪  initWithAttributionCatalog:${new ObjC.Object(args[2])}
     ↪  cameraAttributionListData:${new ObjC.Object(args[3])}]`);
    // Empty Attribution Catalog and List
    args[2] = new ObjC.Object(new NativePointer(0));
    args[3] = new ObjC.Object(new NativePointer(0));
  },

  onLeave(log, retval, state) {
  }
});
```

Listing 7.26: FRIDA script for disabling the recording indicator.

The script in Listing 7.26 targets the attribution catalog, which is used to render the recording indicator. It overwrites the `STMutableMediaStatusDomainData initWithAttributionCatalog:cameraAttributionListData:` method, which was found to be invoked when the camera or microphone is accessed. This disablement of the recording indicator is achieved by nulling the `AttributionCatalog` catalog passed to the function by overwriting it with a `nil` object. Using the approach highlighted above, we successfully disabled the recording indicator.

7.11.1.3 Implications

Whilst the recording indicator serves as a functioning indicator indicating sensor (microphone, camera) usage to the user, the initial implementation was easily circumventable. As shown in Section 7.11.1.2, the iOS component responsible for the recording indicator is the user-space daemon `mediaserverd`. A capable attacker may be able to reach code execution within `mediaserverd` to take over the recording indicator, as it has various Cross-Process Communication (XPC) interfaces reachable from sandboxed apps.[16]

In conclusion, the initial recording indicator is only capable of improving user privacy in scenarios where apps trigger microphone and camera recordings over the intended

[16] Irrespective of Exclaves, Apple limited this risk by splitting `mediaserverd` into multiple, less-privileged daemons as of iOS 17.

frameworks. On a compromised device, the recording indicator is part of a highly exposed system daemon, making successful attacks likely.

7.11.2 Secure Indicator Light

With the shift towards Exclaves, a new mechanism for rendering the recording indicator, now SIL, emerged. When enumerating Exclave resources (see Section 7.4), we find a `com.apple.service.ExclaveIndicatorController` resource residing in the `com.apple.kernel` domain. Furthermore, the ExclaveKit DMG framework contains a `SILManagerComponent.framework`.

We find various references to the Exclave Indicator Controller (EIC) in `osfmk/kern/exclaves_sensors`, where it serves as a management component for sensor control and corresponding memory buffer usage (e.g., it appears to be responsible for starting and stopping sensors). We furthermore find the following function signature with comments depicted in Listing 7.27 in `osfmk/kern/exclaves.h`.

```
/*!
 * @function exclaves_sensor_start
 *
 * @abstract
 * Start accessing a sensor and cause any indicators to display.
 *
 * If multiple clients start the same sensor, the sensor will only
 * actually start on the first client.
 *
 * @param sensor_port
 * A sensor buffer port name returned from exclaves_sensor_create()
 * for the sensor.
 *
 * @param flags to pass to the implementation. Must be 0 for now.
 *
 * @param sensor_status
 * Out parameter filled with the sensor status.
 *
 * @result
 * KERN_SUCCESS or mach system call error code.
 */
SPI_AVAILABLE(macos(14.4), ios(17.4), tvos(17.4), watchos(10.4))
kern_return_t
exclaves_sensor_start(mach_port_t sensor_port, uint64_t flags,
    exclaves_sensor_status_t *sensor_status);
```

Listing 7.27: The `exclaves_sensor_start` signature and comments, taken from `osfmk/kern/exclaves.h`.

From the above listing, we see a direct correlation between the call to `exclaves_sensor_create`, which is, for example, invoked in the control flow of receiving a sensor start invocation from userspace via `exclaves_control_trap`. Due to the redacted code in the further handling of the call, we are unable to determine the exact mechanisms regarding SIL rendering.

Assumed Underlying Structure We can make assumptions based on our current knowledge. In the `com.apple.darwin` component, we find the resource `com.apple.service.SecureRTBuddyDCP`. *RTBuddy* is a term coined by the Apple RTKit, its real-time operating system, serving special use cases where standard operating systems would not suffice. Exemplary use cases are the Display Coprocessor (DCP) and ANE [53]. Based on repeated usage of the `SecureRT` term in Exclave resources located in the Darwin domain, we assume that Apple introduced an `RTKit` Exclave equivalent for performing real-time tasks in the guarded world. The suffix DCP strongly hints towards a display coprocessor-related resource. Based on these findings, we can speculate that the `SILManagerComponent` might directly communicate the need to display the SIL to the DCP via the secure world `RTKit`. This implies that the decision for this rendering and the communication to the relevant coprocessor occur fully within guarded levels and are therefore significantly harder to tamper with.

8 Trusted Execution Monitor

8.1 TXM Fundamentals

TXM is a guarded level component running in GL0. As previously shown, TXM has its own SPTM domain, featuring a variety of frame types (see Section A.11). TXM is responsible for enforcing system policies governing code execution [45]. TXM can be called into via SPTM by calling int `TXM_DOMAIN`. XNU performs this task to facilitate various code signing and entitlement management operations. TXM itself communicates with SPTM via SVCs, and sets up the required dispatching structure and for memory frame retyping. We find TXM to be responsible for code signing and entitlement enforcement.

As in this work, we have decided to focus on the inner workings of SPTM and the recently released Exclaves. We have chosen not to delve deeply into the workings of TXM, but instead to examine its interaction with SPTM and XNU. A more detailed analysis of TXM is left up for future work. We find TXM to be responsible for code signing and entitlement enforcement.

8.2 SPTM Calls

We have previously discovered SPTM calls from TXM (see Section 5.3.3). We can map them according to the known SPTM dispatch structure for applicable calls. For standard SPTM calls calling into its dispatching logic, we found SVC #0 to be used. The result can be seen in Table 8.1.

Dispatch Table	Endpoint ID	SPTM Function
`TXM_BOOTSTRAP`	1	`type_TXM-rx_region_as_TXM_rw_type`
`TXM_BOOTSTRAP`	2	`register_dispatch_table`
`TXM_BOOTSTRAP`	3	`retype`
`TXM_BOOTSTRAP`	4	Unknown as of now. Thread identifying information read.
`RETURN_TO_CALLER`	-	Control Function
`PANIC`	-	Control Function
`EXCEPTION_STATE_SAVED`	-	Control Function

Table 8.1: SPTM dispatch calls for TXM.

Comparable to SK, we find that TXM registers its own dispatch tables for calls directed towards the `TXM_DOMAIN`. It can also employ SPTM to retype memory frames, and further calls into a SPTM function `type_TXM-rx_region_as_TXM_rw_type`. TXM further calls into SPTM using SVC #38 to disable all interrupts and SVC #37 to enable all interrupts, respectively.

8.2.1 Dispatch Table Registration

We find that TXM performs dispatch table registration calls in its entry function. The registration process is very comparable to SK's dispatch table registration. TXM also registers two dispatch tables with different dispatch functions into SPTM. We find the

first dispatch function `TXM_dispatch_function_0` to be registered with permissions allowing XNU to call into it, and the second function `TXM_dispatch_function_1` to be registered with permissions allowing SPTM to call into it. The registration can be seen below in Listing 8.1. It is performed from the TXM entry.

```
register_dispatch_table(0,TXM_dispatch_function_0,2);
register_dispatch_table(1,TXM_disptach_function_1,1);
```

Listing 8.1: Calls to `sptm_register_dispatch_table` from TXM function `FUN_fffffff01701bd60`. TXM registers two dispatching functions that allow calling into TXM when calling the domain `TXM_DOMAIN`. The source code was disassembled by Ghidra.

8.2.2 TXM-`rx` Region Retyping

A further action performed in the TXM entry is an SPTM call to `type_TXM-rx_region_as_TXM_rw_type`. We found this SPTM function to call `type_region`, a function that iterates over a named memory region and retypes all included memory frames to the provided type. The exact reason for this call has not yet been discovered, but we assume it performs TXM lockdown on early boot executable code.

8.3 Calling into TXM

We know from our previous analysis of SPTM that XNU calls into TXM via `txm_enter`, which parameterizes the called domain to `TXM_DOMAIN`, sets the target dispatch table to zero, and conditionally sets the endpoint ID based on input parameters. The main XNU call wrapping this actual SPTM call for invoking TXM functionality shows to be `txm_kernel_call`. The call is parameterized with an input structure of type `txm_call_t`. The type definition can be seen Listing 8.2.

```
typedef struct _txm_call {
    /* Input arguments */
    TXMKernelSelector_t selector;
    TXMReturnCode_t failure_code_silent;
    bool failure_fatal;
    bool failure_silent;
    bool skip_logs;
    uint32_t num_input_args;
    uint32_t num_output_args;

    /* Output arguments */
    TXMReturn_t txm_ret;
    uint64_t num_return_words;
    uint64_t return_words[kTXMStackReturnWords];
} txm_call_t;
```

Listing 8.2: `txm_call_t` type definition in `bsd/sys/trusted_execution.h`.

The selector field acts as the SPTM call endpoint selector. For calls to TXM, a variable number of arguments can be provided, depending on the endpoint called into. Arguments are found to be passed to TXM via thread stacks of type `txm_thread_stack_t`. This custom structure is shown in Listing 8.3.

```
typedef struct _txm_thread_stack {
    /* Virtual mapping of the thread stack page */
    uintptr_t thread_stack_papt;

    /* Physical page used for the thread stack */
    uintptr_t thread_stack_phys;

    /* Pointer to the thread stack structure on the thread stack page
     ↪  */
    TXMThreadStack_t *thread_stack_data;

    /* Linkage for the singly-linked-list */
    SLIST_ENTRY(_txm_thread_stack) link;
} txm_thread_stack_t;
```

Listing 8.3: `txm_thread_t` type definition in `bsd/sys/trusted_execution.h`.

The actual argument to the `txm_enter` call is found to be a pointer to a `sptm_call_regs_t` structure, which may hold up to eight arguments. The first argument is always set to the TXM thread stack's physical address, while the other arguments are set based on the TXM function to invoke and the provided input parameters.

8.3.1 Selector Enumeration

We can enumerate known TXM selectors from the XNU open-source code, together with the number of input and output arguments. The full list is shown in Table 8.2.

Selector Value	#Input	#Output
kTXMKernelSelectorGetTrustCacheInfo	0	4
kTXMKernelSelectorLoadTrustCache	7	0
kTXMKernelSelectorQueryTrustCache	2	2
kTXMKernelSelectorCheckTrustCacheRuntimeForUUID	1	0
kTXMKernelSelectorAddFreeListPage	1	0
kTXMKernelSelectorGetLogInfo	0	3
kTXMKernelSelectorGetCodeSigningInfo	0	6
kTXMKernelSelectorSetSharedRegionBaseAddress	2	0
kTXMKernelSelectorEnterLockdownMode	0	0
kTXMKernelSelectorDeveloperModeToggle	1	0
kTXMKernelSelectorRegisterProvisioningProfile	2	1
kTXMKernelSelectorUnregisterProvisioningProfile	1	2
kTXMKernelSelectorAssociateProvisioningProfile	2	0
kTXMKernelSelectorDisassociateProvisioningProfile	1	0
kTXMKernelSelectorAuthorizeCompilationServiceCDHash	1	0
kTXMKernelSelectorMatchCompilationServiceCDHash	1	1
kTXMKernelSelectorSetLocalSigningPublicKey	1	0
kTXMKernelSelectorGetLocalSigningPublicKey	0	1
kTXMKernelSelectorAuthorizeLocalSigningCDHash	1	0
kTXMKernelSelectorRegisterCodeSignature	3	2
kTXMKernelSelectorUnregisterCodeSignature	1	2

Selector Value	#Input	#Out
kTXMKernelSelectorValidateCodeSignature	1	0
kTXMKernelSelectorReconstituteCodeSignature	1	2
kTXMKernelSelectorRegisterAddressSpace	2	1
kTXMKernelSelectorUnregisterAddressSpace	1	0
kTXMKernelSelectorAssociateCodeSignature	5	0
kTXMKernelSelectorAllowJITRegion	1	0
kTXMKernelSelectorAssociateJITRegion	3	0
kTXMKernelSelectorAllowInvalidCode	1	0
kTXMKernelSelectorAcquireSigningIdentifier	1	1
kTXMKernelSelectorAssociateKernelEntitlements	2	0
kTXMKernelSelectorResolveKernelEntitlementsAddressSpace	1	1
kTXMKernelSelectorAccelerateEntitlements	1	1
kTXMKernelSelectorImage4SetNonce	2	0
kTXMKernelSelectorImage4RollNonce	1	0
kTXMKernelSelectorImage4GetNonce	1	1
kTXMKernelSelectorImage4GetExports	0	1
kTXMKernelSelectorImage4SetReleaseType	1	0
kTXMKernelSelectorImage4SetBootNonceShadow	1	0
kTXMKernelSelectorImage4Dispatch	5	0

Table 8.2: TXM kernel call selectors and argument counts extracted from XNU open source.

The control flow of calling into TXM from XNU with a specific selector is shown below in Figure 8.1.

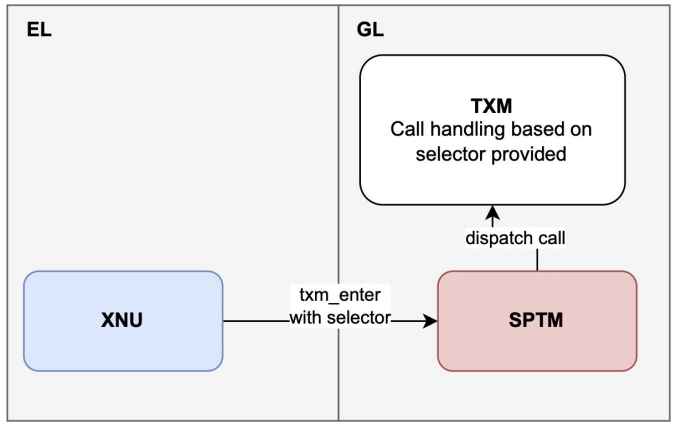


Figure 8.1: Control flow of XNU issuing a request to TXM via a call into SPTM, which dispatches the call to TXM.

8.3.2 TXM Responsibilities

From the selectors available to the SPTM-TXM call and its integration within the XNU open-source code, we know TXM to be the code-signing component in modern Apple devices that supports SPTM and TXM. In XNU open source `bsd/kern/code_signing` directory we find `ppl.c`, `xnu.c`, and `txm.c`. These are the three different code signing monitors XNU employs, with `xnu.c` indicating XNU itself to handle code signing for devices not supporting PPL, `PPL.c` handling code signing on devices running PPL, and now TXM taking over for supported devices [26].

In such a capability, it is TXM's responsibility to provide signature and entitlement services for XNU. For this analysis of guarded level components, we have deemed the complete analysis of the Apple code-signing ecosystem out of scope.

8.3.3 TXM Call Handling

TXM is called from SPTM by invoking its registered dispatching functionality. For calls from XNU, the called TXM dispatch function is `TXM_dispatch_function_1`. We find it to confirm the validity of the caller-provided TXM stack, and then call the `txm_dispatch_handler` function. This function performs a switch on the provided selector value, and then invokes the actual handler function. The full listing of `txm_dispatch_handler` can be found in Section A.22.

8.4 TXM Security

In our previous analysis of SPTM, we have found TXM to own a variety of different SPTM frame types. Its memory is therefore protected from unwanted tampering from XNU in case of a kernel compromise by the fact that XNU is not allowed to map memory in TXM-owned frames (see Listing 5.9). With TXM running in GXF, the underlying access and execute permissions directly protect mapped TXM memory from manipulation by normal exception levels, and all alterations of these permissions must be performed through SPTM, as the sole controlling and management instance for page table manipulation. For a more precise analysis of the underlying permissions encoded by SPRR, an up-to-date analysis of SPRR indices is required to be performed on a system supporting SPTM. Such an analysis was not performed by us and is left open as future work.

9 Discussion

9.1 SPTM Security Implications

The past analysis has shed light on the beginning implementation of a microkernel-like structure in XNU. With the introduction of SPTM, Apple has introduced a new root of trust, with the sole authority regarding frame retyping and mapping. We find SPTM to adhere to a strict rule set on allowed and disallowed memory retyping and page mapping operations. Furthermore, SPTM shows how to realize so-called SPTM domains. SPTM domains serve as privilege boundaries, essentially splitting the XNU kernel up into subsections with different privileges. By doing so, SPTM significantly limits XNU's ability to interact with security-relevant processes. Whilst we have uncovered the fundamental building blocks within SPTM regarding memory security, we have omitted significant parts of the remapping logic. A more detailed analysis would provide a more substantial claim to support the fundamental, underlying security assumptions.

9.2 Exclaves Security Implications

We find Exclaves to isolate a core subset of system functionality from XNU, significantly limiting its ability to interact with it. Whilst XNU can invoke services on Exclaves, as it needs to for proper system function, the actual code and memory are guarded via SPTM by residing in a different SPTM domain in memory frames of non-shared types. This significantly limits the accessible interface to these components in the event of a kernel compromise, as XNU is no longer able to map memory into them nearly arbitrarily. With more and more parts of the system moving into compartmentalization, it appears the progress towards a true microkernel architecture has been kicked off.

9.3 Limitations

We were successful in providing an overview of the new Apple system design, particularly in terms of communication across components, but there are obvious limitations.

Despite the XNU open-source code containing partial information regarding Exclaves, we still lack source code for the entire secure world system. As we had to rely entirely on manual reverse engineering of substantial system components, the analysis became very complex. Whilst we were able to infer general information, especially regarding component interaction, the exact inner workings are still unknown. The analysis was further complicated by the nearly complete lack of official sources regarding SPTM, which led to us having to operate under non-confirmed assumptions about its workings.

Furthermore, as both SPTM and Exclaves have been released very recently, they are still subject to constant and significant change. While it is possible to perform an analysis on a snapshot of current system specifications, new releases may significantly alter behavior and necessitate new reverse engineering.

9.4 Future Work

The analysis of XNU kernel restructuring is far from done with this work. Even though we were able to shed light on the fundamental underlying concepts, especially concerning SPTM rule sets, the restructuring Apple appears to perform is highly significant. Furthermore, large parts of the architecture have yet to be reverse-engineered. The question of request dispatching with regards to TXM calling into SPTM with SVCs whilst other GL0 components appear to invoke them to call into SK remains. We assume that underlying thread scheduling and hypervisor configuration are responsible for this handling, but have not yet delved into this. An additional key aspect that has found little focus in our analysis is the analysis of further GL0 *Exclavecore* components. Their interaction and responsibility in the Exclaves ecosystem is entirely unclear as of now, and should be the focus of future research. The same holds for the early boot process of Exclaves, which we have deemed mainly out of scope for this analysis, and the inclusion of what appears to be a seL4-like microkernel operating in the secure world. We have provided an initial look into the secure kernel, yet a complete analysis remains to be performed.

For Exclaves, we shed light on the underlying transport mechanisms, but there is still much to be discovered. The exact Tightbeam implementation and usage from Exclave components are deemed an interesting target concerning system security. As research on SPTM remains sparse, our analysis aimed to provide an architectural overview rather than conducting a comprehensive security analysis. Such an analysis, combined with a well-founded threat model, is key to evaluating SPTM-based security. Finally, as monolithic kernel architectures are often used for their superior performance, the question of performance drawbacks resulting from the implemented compartmentalization arises and should be examined in future research.

10 Conclusion

In this work, we examined SPTM, its callers, request dispatching logic, and provided functionality. We find XNU to call into SPTM from normal exception levels, and both TXM and SK calling into SPTM from guarded levels. SPTM calls are parameterized via a call of SPTM to be the sole component responsible for page table manipulation and underlying memory frame retyping. It employs a rigid rule set of allowed transitions to limit the caller's ability regarding page table mapping. Via this, SPTM introduces the concept of `SPTM_DOMAINS`, which are realized as trust domains, scoping different components from each other concerning memory accesses. We assume a gain in system security based on these changes, as kernel compromises no longer impact the entire system.

We have furthermore looked into SK, a GL1 component that is part of the *Exclavecore*, and analyzed it in terms of its own request handling capabilities and the requests it performs in turn for SPTM. We find it to act as a kernel managing GL0 components. To serve this purpose, SK is callable from XNU via SPTM, to allow for the scheduling of threads and requests into secure world components. It acts as the privileged management component for its GL0 clients in *Exclavecore*, and handles issuing frame retyping requests to SPTM on their behalf.

The newest addition to Apple's memory protection mechanisms are Exclaves, which serve as groupings of isolated, sensitive services and resources that require special privileges to call into, and are therefore guarded against unwanted tampering from XNU. Scoped capabilities include sensor accesses and the underlying buffer management and ANE services. Calling from XNU into Exclaves revealed a significant amount of complexity surrounding seL4 IPC buffers, which suggests that Apple may be implementing the seL4 microkernel with SK. We managed to interact with Exclaves from the userspace by invoking the provided functionalities from a suitably privileged task, but were unsuccessful in retrieving actual sensor data from an Exclave.

Finally, we provided a brief overview of TXM, which resides in its own *SPTM* domain and is responsible for, among other things, code signing and entitlement verification. These highly sensitive services are therefore scoped away from XNU via SPTM.

In conclusion, we see a clear path towards a more compartmentalized microkernel approach finding its way into Apple's operating systems. The implemented features enhance system security and protect against a total loss of control in the event of a kernel compromise. However, large parts of these mitigations remain unknown, which makes it very difficult to make reliable statements regarding the underlying security assumptions. With Exclaves as a rapidly evolving security enhancement, changes to the system structure in the near future are to be expected and may once again completely alter the system architecture.

Bibliography

[1] National Security Agency (NSA). *Ghidra Software Reverse Engineering Framework*. 2025. URL: https://github.com/NationalSecurityAgency/ghidra (cited on page 16).

[2] Apple Inc. *Creating Dynamic Libraries*. https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/CreatingDynamicLibraries.html. 2012 (cited on page 66).

[3] Apple Inc. *macOS Sequoia 15.5 Release Notes*. https://developer.apple.com/documentation/macos-release-notes/macos-15_5-release-notes. Apple Developer Documentation. May 2025 (cited on pages 26, 115 sq., 132).

[4] Apple Documentation Archive. *Mach*. 2013. URL: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html (cited on page 68).

[5] ARM. *ERET*. 2020. URL: https://developer.arm.com/documentation/ddi0602/2024-03/Base-Instructions/ERET--Exception-Return- (cited on page 54).

[6] ARM. *ESR_EL1, Exception Syndrome Register (EL1)*. 2020. URL: https://developer.arm.com/documentation/ddi0595/2020-12/AArch64-Registers/ESR-EL1--Exception-Syndrome-Register--EL1- (cited on page 34).

[7] ARM. *SPSR - Saved Program Status Register*. 2020. URL: https://developer.arm.com/documentation/ddi0601/2025-06/AArch32-Registers/SPSR--Saved-Program-Status-Register (cited on page 38).

[8] ARM. *Exception level*. 2024. URL: https://developer.arm.com/documentation/102412/0103/Privilege-and-Exception-levels/Exception-levels (cited on page 9).

[9] ARM. *Armv8-A Address Translation*. 2025. URL: https://developer.arm.com/documentation/100940/latest/ (cited on pages 10, 51).

[10] ARM. *Exception Handling*. 2025. URL: https://developer.arm.com/documentation/den0013/d/Exception-Handling (cited on page 9).

[11] ARM. *HCR_EL2, Hypervisor Configuration Register*. 2025. URL: https://developer.arm.com/documentation/ddi0601/2025-06/AArch64-Registers/HCR-EL2--Hypervisor-Configuration-Register (cited on page 36).

[12] ARM. *Multilevel translation*. 2025. URL: https://developer.arm.com/documentation/101811/0104/The-Memory-Management-Unit--MMU-/Multilevel-translation (cited on page 10).

[13] ARM. *Permission attributes*. 2025. URL: https://developer.arm.com/documentation/102376/0100/Permissions-attributes (cited on page 11).

[14] ARM. *Registers, vectors, lanes and elements*. 2025. URL: https://developer.arm.com/documentation/102474/0100/Fundamentals-of-Armv8-Neon-technology/Registers--vectors--lanes-and-elements (cited on page 26).

[15] ARM. *Supervisor Call - SVC*. 2025. URL: https://developer.arm.com/documentation/107706/0100/System-exceptions/Supervisor-Call---SVC (cited on page 9).

[16] ARM. *Taking an exception*. 2025. URL: https://developer.arm.com/documentation/102412/0103/Handling-exceptions/Taking-an-exception (cited on pages 21, 36).

[17] ARM. *Types of privilege*. 2025. URL: https://developer.arm.com/documentation/102412/0103/Privilege-and-Exception-levels/Types-of-privilege (cited on page 10).

[18] ARM. *VBAR_EL1, Vector Base Address Register (EL1)*. 2025. URL: https://developer.arm.com/documentation/ddi0601/2025-03/AArch64-Registers/VBAR-EL1--Vector-Base-Address-Register--EL1- (cited on pages 21, 36, 50).

[19] ARM. *Virtual and physical addresses*. 2025. URL: https://developer.arm.com/documentation/101811/0104/Virtual-and-physical-addresses (cited on page 10).

[20] Asahi Linux. *m1n1: an experimentation playground for Apple Silicon*. 2024. URL: https://github.com/AsahiLinux/m1n1/blob/main/tools/apple_regs.json (cited on pages 17, 21).

[21] Random Augustine. *On Apple Exclaves*. 2025. URL: https://randomaugustine.medium.com/on-apple-exclaves-d683a2c37194 (cited on pages 7 sq., 14, 67).

[22] Brandon Azad. *The core of Apple is PPL: Breaking the XNU kernel's kernel*. https://googleprojectzero.blogspot.com/2020/07/. Accessed: 2025-07-11. July 2020 (cited on page 43).

[23] Blacktop. *ipsw*. 2025. URL: https://github.com/blacktop/ipsw (cited on page 16).

[24] Blacktop. *ipsw symbolication signatures*. 2025. URL: https://github.com/blacktop/symbolicator (cited on page 16).

[25] Dataflow Forensics. *iOS 17: New Version, New Acronyms*. 2023. URL: https://www.df-f.com/blog/ios17 (cited on pages 8, 36 sq.).

[26] Dataflow Forensics. *iOS 17: New Version, New Acronyms | Round 2*. 2023. URL: https://www.df-f.com/blog/ios-17round2 (cited on pages 8, 22, 24, 104).

[27] Dataflow Forensics. *Tracing Back to the Source | SPTM Round 3*. 2025. URL: https://www.df-f.com/blog/sptm3 (cited on page 8).

[28] Asahi Linux Documentation. *SPRR and GXF*. 2025. URL: https://asahilinux.org/docs/hw/cpu/sprr-gxf/ (cited on page 12).

[29] The Apple Wiki - Community Forum. *Kernel Syscalls*. 2020. URL: https://theapplewiki.com/wiki/Kernel_Syscalls (cited on page 64).

[30] FRIDA. *FRIDA - Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*. 2025. URL: https://frida.re (cited on page 98).

[31] Gernot Heiser. *The seL4 Microkernel – An Introduction*. White paper Revision 1.4. Revision 1.4 of 2025-01-08. The seL4 Foundation, Jan. 2025 (cited on page 71).

[32] Apple Inc. *XNU*. 2024. URL: https://github.com/apple-oss-distributions/xnu/tree/rel/xnu-10063 (cited on pages 15, 58, 138, 143, 145).

[33] Apple Inc. *Apple Security Research Device Program*. 2025. URL: https://security.apple.com/research-device/ (cited on page 15).

[34] Apple Inc. *xnu*. 2025. URL: https://github.com/apple-oss-distributions/xnu (cited on pages 7, 15).

[35] Jonathan Levin. *Casa De P(a)P(e)L - Explaining Apple's Page Protections Layer in A12 CPUs*. 2019. URL: https://newosxbook.com/articles/CasaDePPL.html (cited on page 12).

[36] Jonathan Levin. *disarm - Quick (& formerly dirty) CLI instruction lookup for ARM64, turned full fledged binary analyzer*. 2025. URL: https://newosxbook.com/tools/disarm.html (cited on page 16).

[37] LLDB Project. *LLDB Debugger*. 2025. URL: https://lldb.llvm.org (cited on page 16).

[38] Robert N. M. Watson Marshall Kirk McKusick George V. Neville-Neil. *The design and implementation of the FreeBSD operating system*. 2nd Edition. Addison-Wesley, 2015, page 45. ISBN: 978-0-321-96897-5 (cited on page 7).

[39] Bojan Novković and Marin Golub. "Improving monolithic kernel security and robustness through intra-kernel sandboxing". In: *Computers & Security* 127 (2023), page 1. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103104 (cited on page 7).

[40] Sven Peter. *Apple Silicon Hardware Secrets: SPRR and Guarded Exception Levels (GXF)*. 2021. URL: https://blog.svenpeter.dev/posts/m1_sprr_gxf/ (cited on pages 11 sqq., 20 sq.).

[41] plzdonthaxme. *accp-h16g-core-sysregs.txt*. 2025. URL: https://gist.github.com/justtryingthingsout/73bf33903d13a0fba12dbac92ee7cd04/ (cited on page 17).

[42] Random Augustine. *Disassembling Apple Exclaves*. 2025. URL: https://randomaugustine.medium.com/disassembling-apple-exclaves-7979bb987f86 (cited on pages 8, 14, 25, 67).

[43] Random Augustine. *More speculation on exclaves*. 2025. URL: https://randomaugustine.medium.com/more-speculation-on-exclaves-d4b94182b54b (cited on pages 8, 14).

[44] Benjamin Roch. "Monolithic kernel vs. Microkernel". In: *TU Wien* 1 (2004), page 1 (cited on page 7).

[45] Apple Platform Security. *Operating system integrity*. 2024. URL: https://support.apple.com/en-gb/guide/security/sec8b776536b/1/web/ (cited on pages 7, 11 sq., 101).

[46] seL4. *Capabilities*. 2020. URL: https://docs.sel4.systems/Tutorials/capabilities.html (cited on page 71).

[47] seL4 Foundation. *seL4 Reference Manual*. Accessed: 2025-07-10, page 18. 2024. URL: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf (cited on page 75).

[48] Amit Singh. *Section 9.3. Mach IPC: The Mac OS X Implementation*. 2006. URL: https://flylib.com/books/en/3.126.1.105/2/ (cited on page 69).

[49] Apple Support. *About iOS 14 Updates*. 2021. URL: https://support.apple.com/en-us/118390 (cited on page 97).

[50] palera1n team. *palera1n - Jailbreak for iPhone, iPad, Macbooks, and AppleTV's for versions 15 and higher*. 2024. URL: https://palera.in (cited on pages 15, 98).

[51] The Apple Wiki contributors. *Filesystem:/System/Library/Frameworks*. Apr. 2025. URL: https://theapplewiki.com/wiki/Filesystem:/System/Library/Frameworks (cited on page 55).

[52] Ian Wienand. *Chapter 6.4:* Physical Memory. https://bottomupcs.com/ch06s04.html. Accessed: 2025-07-11. Computer Science from the Bottom Up, 2004–2022 (cited on page 39).

[53] The Apple Wiki. *RTKit*. https://theapplewiki.com/wiki/RTKit. 2024 (cited on page 100).

[54] Google Project Zero. *Examining Pointer Authentication on the iPhone XS*. 2025. URL: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html (cited on page 11).

A Appendix

A.1 Symbol to Address Mapping for Analyzed Binaries

Table A.1: Symbols encountered and labeled during the reverse engineering of this work, grouped by binary.

Binary	Symbol	Address
Binary: SPTM		
SPTM	gxf_setup_early	fffffff02708b8e8
SPTM	gxf_setup_late	fffffff02708b918
SPTM	gxf_entry_point	fffffff02708053c
SPTM	CORE_DISPATCH_STRUCTURE_POINTER	fffffff027079500
SPTM	Function_Table_Structure	fffffff027079508
SPTM	init_xnu_ro_data	fffffff0270987b4
SPTM	IOMMU_bootstrap	fffffff0270be298
SPTM	register_iommu	fffffff0270bf164
SPTM	CORE_SPTM_FUNCTION	fffffff0270bec68
SPTM	SPECIAL_DISPATCH_STRUCTURE	fffffff027078d80
SPTM	low_level_logger	fffffff0270a029c
SPTM	genter_dispatch_entry	fffffff0270bf3f8
SPTM	synchronous_exception_handler_from_lower	fffffff027081e38
SPTM	retype_frames	fffffff0270b4118
SPTM	retype	fffffff0270c4ad8
SPTM	SPTM_Retype_Global_Table	fffffff027019120
SPTM	RETYPE_Typeout_Structure	fffffff027019150
SPTM	RETYPE_Flag_Structure	fffffff027019140
SPTM	PAPT_permission_update	fffffff0270b2a38
SPTM	SPRR_Index_From_Type	fffffff027019114
SPTM	sk_types_retype_out	fffffff0270ba318
SPTM	map_page	fffffff0270c51d4
SPTM	table_sptm_type	fffffff0270c51d4
SPTM	REMAP_STRUCTURE	fffffff027019130
SPTM	NVME_DISPATCH_TABLE	fffffff027014710
SPTM	SK_BOOTSTRAP_TABLE	fffffff027018c00
SPTM	t8110DART_DISPATCH_TABLE	fffffff027014c30
SPTM	CPUTRACE_DISPATCH_TABLE	fffffff027014490
SPTM	UAT_DISPATCH_TABLE	fffffff0270140c0
SPTM	SART_DISPATCH_TABLE	fffffff0270149b0
SPTM	sptm_dispatch	fffffff0270bf268
SPTM	XNU_BOOTSTRAP_TABLE	fffffff0270186d8
SPTM	TXM_BOOTSTRAP_TABLE	fffffff027018980
SPTM	AllowedCallerDomains	fffffff027019100
SPTM	tb_endpoint_create_with_value	fffffff0088ee9ec
SPTM	tb_client_connection_create_with_endpoint	fffffff0088ed99c

Binary	Symbol	Address
SPTM	__tb_connection_create_transport_for_endpoint	fffffff00088ed624
SPTM	tb_xnu_transport_create	fffffff00088f0eec
SPTM	tb_xnu_transport_send_message	fffffff00088f0fa4
SPTM	__tb_connection_message_construct	fffffff00088ee154
SPTM	_tb_connection_send_query	fffffff00088eda7c
SPTM	exclaves_xnu_proxy_endpoint_call	fffffff008148028
Binary: XNU		
XNU	GENTER_main_gate	fffffff00ab3510c
XNU	genter_setup	fffffff0080f933c
XNU	txm_enter	fffffff00088eb384
XNU	txm_kernel_call	fffffff00863e048
XNU	exclaves_enter	fffffff00813b844
XNU	sk_enter	fffffff00088eb330
XNU	exclaves_bootinfo	fffffff00813b7c8
XNU	sptm_retype	fffffff00088ebc0c
XNU	sptm_map_page	fffffff00088ebc24
XNU	tb_endpoint_create_with_value	fffffff00088ee9ec
XNU	allocate_in_structure	fffffff00816b4a4
XNU	xnu_tb_connection_create	fffffff00088ed674
XNU	__tb_connection_create_transport_for_endpoint	fffffff00088ed624
XNU	xnu_tb_endpoint_destruct	fffffff00088eea5c
XNU	__tb_connection_create_transport_for_endpoint	fffffff00088f10a0
XNU	tb_xnu_transport_create	fffffff00088f1228
XNU	tb_xnu_client_transport_vtable	fffffff007c17938
XNU	xnu_tb_connection_create.	fffffff00088ed674
XNU	tb_client_connection_activate	fffffff00088eda30
XNU	activate_connection	fffffff00088ee6d4
XNU	__tb_connection_message_construct	fffffff00088ee154
XNU	tb_transport_construct_message_buffer	fffffff00088ee73c
XNU	tb_connection_send_query	fffffff00088eda7c
XNU	tb_xnu_transport_send_message.	fffffff00088f0fa4
XNU	make_xnuproxy_endpoint_call_from_tightbeam	fffffff00813af30
Binary: SK		
SK	main_sptm_gate	ffffff8000001784
SK	special_sptm_gate	ffffff800000178c
SK	register_dispatch_table	ffffff8000004340
SK	retype	ffffff80000043b8
SK	get_frame_type	ffffff80000043a0
SK	sptm_t8110_dart_map_table	ffffff8000004310
SK	sptm_t8110_dart_unmap_table	ffffff8000004328
SK	retype_to_shared1	ffffff8000004d44
SK	retype_to_shared1	ffffff8000006ed4
SK	map_to_sk_default1	ffffff8000004dd4
SK	retype_to_shared2	ffffff8000006ed4
SK	retype_to_sk_default1	ffffff8000004dd4
SK	retype_to_XNU_default	ffffff8000006f34
SK	retype_to_sk_default2	ffffff8000007288
SK	SVC_handler_from_lower_EL	ffffff8000008264
SK	ExceptionHandlerBase	ffffff8000003800
SK	RestoreContextAndEret	ffffff80000040a4
SK	FunctionTable	ffffff8000012480

Binary	Symbol	Address
SK	SPTM_dispatch_function_0	ffffff8000001858
SK	SPTM_dispatch_function_1	ffffff80000018b8
SK	SK_called_from_sptm_dispatch_function_0	ffffff80000066e8
Binary: TXM		
TXM	core_SPTM_gate	fffffff01705c058
TXM	type_TXM-rx_region_as_TXM_rw_type	fffffff01705c070
TXM	register_dispatch_table	fffffff017043edc
TXM	retype	fffffff017043ec4
TXM	TXM_dispatch_function_0	fffffff017024c6c
TXM	TXM_disptach_function_1	fffffff017024b00
TXM	txm_dispatch_handler	fffffff017022cb0

A.2 SPTM Domains IDs

Domain	ID
SPTM_DOMAIN	0
XNU_DOMAIN	1
TXM_DOMAIN	2
SK_DOMAIN	3
XNU_HIB_DOMAIN	4
MAX_DOMAINS	5

Table A.2: SPTM domain IDs retrieved from the `sptm_common.h` header file [3].

A.3 SPTM Dispatch Table IDs

Dispatch Table	ID
SPTM_DISPATCH_TABLE_XNU_BOOTSTRAP	0
SPTM_DISPATCH_TABLE_TXM_BOOTSTRAP	1
SPTM_DISPATCH_TABLE_SK_BOOTSTRAP	2
SPTM_DISPATCH_TABLE_T8110_DART_XNU	3
SPTM_DISPATCH_TABLE_T8110_DART_SK	4
SPTM_DISPATCH_TABLE_SART	5
SPTM_DISPATCH_TABLE_NVME	6
SPTM_DISPATCH_TABLE_UAT	7
SPTM_DISPATCH_TABLE_SHART	8
SPTM_DISPATCH_TABLE_RESERVED	9
SPTM_DISPATCH_TABLE_HIB	10
SPTM_DISPATCH_TABLE_INVALID	11

Table A.3: SPTM dispatch table IDs retrieved from the `sptm_common.h` header file [3].

A.4 SPTM Endpoint IDs

Function	ID
SPTM_FUNCTIONID_LOCKDOWN	0
SPTM_FUNCTIONID_RETYPE	1
SPTM_FUNCTIONID_MAP_PAGE	2
SPTM_FUNCTIONID_MAP_TABLE	3
SPTM_FUNCTIONID_UNMAP_TABLE	4

Function	ID
SPTM_FUNCTIONID_UPDATE_REGION	5
SPTM_FUNCTIONID_UPDATE_DISJOINT	6
SPTM_FUNCTIONID_UNMAP_REGION	7
SPTM_FUNCTIONID_UNMAP_DISJOINT	8
SPTM_FUNCTIONID_CONFIGURE_SHAREDREGION	9
SPTM_FUNCTIONID_NEST_REGION	10
SPTM_FUNCTIONID_UNNEST_REGION	11
SPTM_FUNCTIONID_CONFIGURE_ROOT	12
SPTM_FUNCTIONID_SWITCH_ROOT	13
SPTM_FUNCTIONID_REGISTER_CPU	14
SPTM_FUNCTIONID_FIXUPS_COMPLETE	15
SPTM_FUNCTIONID_SIGN_USER_POINTER	16
SPTM_FUNCTIONID_AUTH_USER_POINTER	17
SPTM_FUNCTIONID_REGISTER_EXC_RETURN	18
SPTM_FUNCTIONID_CPU_ID	19
SPTM_FUNCTIONID_SLIDE_REGION	20
SPTM_FUNCTIONID_UPDATE_DISJOINT_MULTIPAGE	21
SPTM_FUNCTIONID_REG_READ	22
SPTM_FUNCTIONID_REG_WRITE	23
SPTM_FUNCTIONID_GUEST_VA_TO_IPA	24
SPTM_FUNCTIONID_GUEST_STAGE1_TLBOP	25
SPTM_FUNCTIONID_GUEST_STAGE2_TLBOP	26
SPTM_FUNCTIONID_GUEST_DISPATCH	27
SPTM_FUNCTIONID_GUEST_EXIT	28
SPTM_FUNCTIONID_MAP_SK_DOMAIN	29
SPTM_FUNCTIONID_HIB_BEGIN	30
SPTM_FUNCTIONID_HIB_VERIFY_HASH_NON_WIRED	31
SPTM_FUNCTIONID_HIB_FINALIZE_NON_WIRED	32
SPTM_FUNCTIONID_IOFILTER_PROTECTED_WRITE	33

Table A.4: SPTM endpoint IDs retrieved from the `sptm_xnu.h` header [3].

A.5 IOMMU IDs

IOMMU	ID
IOMMU_ID_SHART	0
IOMMU_ID_SART	1
IOMMU_ID_NVME	2
IOMMU_ID_UAT	3
IOMMU_ID_DART_T8020	4
IOMMU_ID_DART_T8110	5
IOMMU_ID_DART_T6000	6

Table A.5: IOMMU IDs retrieved from the `sptm_xnu.h` header [3].

A.6 `IOMMU_bootstrap` Function in SPTM

```
void IOMMU_bootstrap(ulong iommu_id?)

{
  uint bootstrap_stage;
  byte bVar1;
  int iVar2;
  undefined8 PAPT_Address;
  ulong iommu_id;
```

```
ulong permissions;
IOMMU_struct *IOMMU_Struct;
long current_GL;
TPIDR *tpidr_e/gL2;
TPIDR *tpidr_el2/gl2;
long gxf_status;
long iommu_id_offset;
byte *valid_flag;

if ((bootstrap_stage >> 7 & 1) == 0) {
                    /* does the iommu id check out compared with the
                     ↪  current state? */
  iommu_id_offset = (iommu_id? & 0xffffffff) * 0x20;
  valid_flag = &LAB_ffffff027078c50 + iommu_id_offset;
                    /* if a functions resided there call it and store
                     ↪  value */
  if (*(code **)(*(long *)(&IOMMUStruct + iommu_id_offset) + 8) ==
   ↪  (code *)0x0) {
    *valid_flag = 1;
  }
  else {
    bVar1 = (**(code **)(*(long *)(&IOMMUStruct + iommu_id_offset) +
     ↪  8))();
    *valid_flag = bVar1;
    if ((bVar1 & 1) == 0) {
      return;
    }
  }
                    /* if we are in the expected boot stage, check if
                     ↪  the iommu is valid */
  if ((bootstrap_stage >> 7 & 1) == 0) {
    if ((*valid_flag & 1) == 0) {
                    /* WARNING: Subroutine does not return */
      cleanup_and_panic("IOMMU with id %d not supported");
    }
    IOMMU_Struct = *(IOMMU_struct **)(&IOMMUStruct +
     ↪  iommu_id_offset);
    if ((bootstrap_stage >> 7 & 1) == 0) {
                    /* GXF Status */
      gxf_status = UnkSytemRegRead(3,6,0xf,8,0);
      if (gxf_status == 0) {
        tpidr_el2/gl2 = (TPIDR *)tpidr_el2;
      }
      else {
        tpidr_el2/gl2 = (TPIDR *)UnkSytemRegRead(3,6,0xf,0xb,1);
      }
      *(ulong *)&tpidr_el2/gl2->dispatchID = (iommu_id? & 0xffffffff)
       ↪  + 1;
      if ((bootstrap_stage >> 7 & 1) == 0) {
        PAPT_Address = reserve_pages_for_IOMMU?
                                  (SPTM_IOMMU_BOOTSTRAP,
                                   (int)IOMMU_Struct->size? * (uint)IO
                                    ↪  MMU_Struct->pages_address? +
                                   0x3fff >> 0xe);
        *(undefined8 *)(&IOMMU_Struct_8c60 + iommu_id_offset) =
         ↪  PAPT_Address;
        *(undefined2 *)(&IOMMU_Struct_8c68 + iommu_id_offset) = 0;
```

```
57            iVar2 = (*(code *)IOMMU_Struct->field16_0x10)();
58                      /* verify we have a dispatch table here */
59            if (IOMMU_Struct->dispatchTable == 0) {
60                      /* WARNING: Subroutine does not return */
61              cleanup_and_panic("IOMMU \"%s\" does not provide an
                 ↪  XNU-facing dispatch table");
62            }
63                      /* if NVME? */
64            if ((int)iommu_id? == 2) {
65              iommu_id = 2;
66              permissions = 0x12;
67            }
68            else {
69              permissions = 2;
70              iommu_id = iommu_id?;
71            }
72                      /* then we call register dispatch table with the
                         ↪  dispatch_table pointer, and
73                        some others
74                        +0x38 has the table */
75            register_iommu(iommu_id,IOMMU_Struct->dispatchTable_id,IOMMU_
               ↪  Struct->dispatchTable,
76                           permissions);
77            if (IOMMU_Struct->dispatch_table != 0) {
78                      /* is this kinda like SK dart table specific? */
79              register_iommu(iommu_id?,IOMMU_Struct->dispatchTable_id +
                 ↪  1,IOMMU_Struct->dispatch_table
80                             ,8);
81            }
82            if ((bootstrap_stage >> 7 & 1) == 0) {
83              current_GL = UnkSytemRegRead(3,6,0xf,8,0);
84              if (current_GL == 0) {
85                tpidr_e/gL2 = (TPIDR *)tpidr_el2;
86              }
87              else {
88                tpidr_e/gL2 = (TPIDR *)UnkSytemRegRead(3,6,0xf,0xb,1);
89              }
90              *(undefined8 *)&tpidr_e/gL2->dispatchID = 9;
91              if (iVar2 == 0) {
92                return;
93              }
94                      /* WARNING: Subroutine does not return */
95              cleanup_and_panic("IOMMU \"%s\" bootstrap failure");
96            }
97          }
98        }
99      }
100   }
101                     /* WARNING: Subroutine does not return */
102   panic?("%s: Unexpected bootstrap stages reached. Expected: %p, Actual:
       ↪  %p");
103 }
```

Listing A.1: `IOMMU_bootstrap` function in SPTM.

A.7 SPTM State Transition Actions

State	Event	Transition Action Label	Next State	Domain	Flag	Domain Name
0x0	0x0	FUN_ffffffff0270bf150	0x2			
0x0	0x1	FUN_ffffffff0270bf13c	0x1			
0x0	0x9	FUN_ffffffff0270816e4	0x13			
0x1	0x2	FUN_ffffffff027080248	0x4	0x3	0x1	SK_DOMAIN
0x1	0x5	FUN_ffffffff0270812ac	0x0			
0x1	0x9	FUN_ffffffff02708173c	0x13	0x3		SK_DOMAIN
0x2	0x2	FUN_ffffffff027015fc0	0x3	0x2	0x1	TXM_DOMAIN
0x2	0x5	FUN_ffffffff027081258	0x0			
0x2	0x9	FUN_ffffffff0270816e4	0x13	0x2		TXM_DOMAIN
0x3	0x5	FUN_ffffffff027080e1c	0x2			
0x3	0x9	FUN_ffffffff0270816e4	0x13			
0x4	0x5	FUN_ffffffff027080eb4	0x1			
0x4	0x9	FUN_ffffffff0270816e4	0x13			
0x5	0x2	FUN_ffffffff027080248	0xb	0x1	0x1	XNU_DOMAIN
0x5	0x3	FUN_ffffffff027080000	0x8	0x1	0x3	XNU_DOMAIN
0x5	0x4	FUN_ffffffff027080100	0x10	0x1	0x1	XNU_DOMAIN
0x5	0xb	FUN_ffffffff027081798	0x15	0x1		XNU_DOMAIN
0x5	0xc	FUN_ffffffff027080248	0x6	0x1	0x1	XNU_DOMAIN
0x5	0xe	FUN_ffffffff027080248	0xc	0x1	0x1	XNU_DOMAIN
0x6	0x5	FUN_ffffffff02708048c	0x6			
0x6	0x6	FUN_ffffffff0270bf098	0x6			
0x6	0x9	FUN_ffffffff027081570	0x13			
0x6	0xb	FUN_ffffffff027081798	0x15			
0x6	0xd	FUN_ffffffff027080248	0xb	0x1	0x1	XNU_DOMAIN
0x7	0x2	FUN_ffffffff027080248	0xc	0x4	0x1	XNU_HIB_DOMAIN
0x7	0xb	FUN_ffffffff027081798	0x15	0x4		XNU_HIB_DOMAIN
0x8	0x2	FUN_ffffffff027080248	0xe	0x2	0x1	TXM_DOMAIN
0x8	0x5	FUN_ffffffff02708048c	0x5	0x2		TXM_DOMAIN
0x8	0x6	FUN_ffffffff027080164	0x9	0x2		TXM_DOMAIN
0x8	0x9	FUN_ffffffff027081570	0x13	0x2		TXM_DOMAIN
0x8	0xb	FUN_ffffffff027081798	0x15	0x2		TXM_DOMAIN
0x9	0x7	FUN_ffffffff0270bf098	0x5	0x2		TXM_DOMAIN
0xa	0x5	FUN_ffffffff027081014	0xb	0x1		XNU_DOMAIN
0xa	0x9	FUN_ffffffff027081570	0x13	0x2		TXM_DOMAIN
0xa	0xb	FUN_ffffffff027081798	0x2			
0xb	0x3	FUN_ffffffff027080000	0xa		0x1	
0xb	0x5	FUN_ffffffff02708048c	0x5			
0xb	0x6	FUN_ffffffff0270bf098	0xd			
0xb	0x9	FUN_ffffffff027081570	0x13			
0xb	0xb	FUN_ffffffff027081798	0x15			
0xc	0x4	FUN_ffffffff027080100	0x11		0x1	
0xc	0x5	FUN_ffffffff02708048c	0x7			
0xc	0x9	FUN_ffffffff027081570	0x13			
0xc	0xb	FUN_ffffffff027081798	0x15			
0xd	0x2	FUN_ffffffff027080308	0x16	0x1	0x1	XNU_DOMAIN
0xd	0x8	FUN_ffffffff027080930	0xb	0x1		XNU_DOMAIN
0xd	0xb	FUN_ffffffff027081798	0x15	0x1		XNU_DOMAIN
0xe	0x5	FUN_ffffffff027080e1c	0x8	0x1		XNU_DOMAIN
0xe	0x9	FUN_ffffffff027081488	0x13			
0xe	0xb	FUN_ffffffff027081798	0x15			
0xf	0x5	FUN_ffffffff027080eb4	0x10	0x1		XNU_DOMAIN
0xf	0x9	FUN_ffffffff0270814d0	0x13			
0xf	0xb	FUN_ffffffff027081798	0x15			
0x10	0x2	FUN_ffffffff027080248	0xf	0x3	0x1	SK_DOMAIN
0x10	0x5	FUN_ffffffff02708048c	0x5	0x3		SK_DOMAIN
0x10	0x7	FUN_ffffffff0270bf098	0x5	0x3		SK_DOMAIN
0x10	0x9	FUN_ffffffff0270814f4	0x13	0x3		SK_DOMAIN
0x10	0xb	FUN_ffffffff027081798	0x15	0x3		SK_DOMAIN

State	Event	Transition Action Label	Next State	Domain	Flag	Domain Name
0x11	0x5	FUN_ffffffo27081014	0xc	0x3		SK_DOMAIN
0x11	0xb	FUN_ffffffo27081798	0x15	0x3		SK_DOMAIN
0x16	0x5	FUN_ffffffo2708048c	0xd			
0x16	0x9	FUN_ffffffo27081570	0x13			
0x16	0xb	FUN_ffffffo27081798	0x15			

Table A.6: The table shows SPTM-internal state transitions, based on a current state and an event. For valid state transitions, a transition action is defined and executed. The predetermined structure further defines the next state to transition into, an SPTM domain field, and a flag, which are used for validity checking.

A.8 `genter_dispatch_entry` Function

```
void genter_dispatch_entry(ulong sptm_dispatch_target,undefined8
 ↪  param_2)
{
  ushort domain_id;
  undefined4 internal_event_id;
  uint endpoint_id;

  domain_id = (ushort)(sptm_dispatch_target >> 0x30) & 0xff;
  if ((sptm_dispatch_target & 0xff000000000000) == 0) {
                    // Domain: SPTM_DOMAIN
    if ((sptm_dispatch_target & 0xff00000000) == 0) {
                    // Dispatch Table:
                    // SPTM_DISPATCH_TABLE_XNU_BOOTSTRAP
      endpoint_id = (uint)sptm_dispatch_target & 0xff;
      if (endpoint_id == 0x1b) {
        internal_event_id = 0xc;
      }
      else {
        if (endpoint_id != 0x1c) {
          internal_event_id = 0xe;
          if (endpoint_id != 0x1e) {
            internal_event_id = 2;
          }
          goto CORE_FUNCTION_CALLER;
        }
        internal_event_id = 0xd;
      }
      if ((global_flag & 1) == 0) {
        debug?(0x56,param_2,"%s(%s:%d) - %s(%#llx), %s(%#llx),
         ↪  %s(%#llx), %s(%#llx), %s(%#llx)\n");
      }
    }
    else {
      internal_event_id = 2;
    }
  }
  else {
        // Domain called not SPTM_DOMAIN
    if (dispatch_table_id == 2) {
        // Domain: TXM_DOMAIN
      internal_event_id = 3;
    }
```

```
    else {
      if (dispatch_table_id != 3) {
        debug?(0x25,param_2,"%s(%s:%d) - %s(%#llx), %s(%#llx),
         ↪  %s(%#llx)\n");
      }
      // Domain: SK_DOMAIN
      internal_event_id = 4;
    }
  }
CORE_FUNCTION_CALLER:
  CORE_SPTM_FUNCTION(internal_event_id,sptm_dispatch_target);
  return;
}
```

Listing A.2: The `genter_dispatch_entry` is function responsible for dispatching `GENTER` calls into SPTM.

A.9 `synchronous_handler_from_lower` SPTM Function

```
void synchronous_exception_handler_from_lower
   (undefined8 param_1,undefined1 param_2 [16],undefined1 param_3 [16],
   undefined1 param_4 [16],undefined1 param_5 [16],undefined1 param_6
    ↪  [16],
   undefined1 param_7 [16],undefined1 param_8 [16],undefined1 param_9
    ↪  [16],
   undefined8 param_10,ulong param_11,undefined8 param_12,undefined8
    ↪  param_13,
   undefined8 param_14,undefined8 param_15,undefined8
    ↪  param_16,undefined8 param_17)

{
  undefined8 uVar1;
  undefined8 uVar2;
  undefined8 uVar3;
  char *pcVar4;
  ulong EC;
  ulong SVC_Imm;
  undefined8 in_x9;
  ulong dispatch_table;
  ulong dispatch_table_id;
  long lVar5;
  undefined8 *puVar6;
  undefined8 in_x10;
  ulong some_flag_from_TPIDR_GL2;
  undefined8 in_x11;
  undefined8 in_x12;
  undefined8 in_x13;
  undefined8 in_x14;
  undefined8 in_x15;
  ulong in_x16;
  undefined8 in_x17;
  undefined8 in_x18;
  undefined8 unaff_x19;
  undefined8 unaff_x20;
```

```
  undefined8 unaff_x21;
  undefined8 unaff_x22;
  undefined8 unaff_x23;
  undefined8 unaff_x24;
  undefined8 unaff_x25;
  undefined8 unaff_x26;
  undefined8 unaff_x27;
  undefined8 unaff_x28;
  undefined8 unaff_x29;
  undefined8 unaff_x30;
  undefined8 unaff_d8;
  undefined8 in_register_00005108;
  undefined8 unaff_d9;
  undefined8 in_register_00005128;
  undefined8 unaff_d10;
  undefined8 in_register_00005148;
  undefined8 unaff_d11;
  undefined8 in_register_00005168;
  undefined8 unaff_d12;
  undefined8 in_register_00005188;
  undefined8 unaff_d13;
  undefined8 in_register_000051a8;
  undefined8 unaff_d14;
  undefined8 in_register_000051c8;
  undefined8 unaff_d15;
  undefined8 in_register_000051e8;
  undefined8 in_d16;
  undefined8 in_register_00005208;
  undefined8 in_d17;
  undefined8 in_register_00005228;
  undefined8 in_d18;
  undefined8 in_register_00005248;
  undefined8 in_d19;
  undefined8 in_register_00005268;
  undefined8 in_d20;
  undefined8 in_register_00005288;
  undefined8 in_d21;
  undefined8 in_register_000052a8;
  undefined8 in_d22;
  undefined8 in_register_000052c8;
  undefined8 in_d23;
  undefined8 in_register_000052e8;
  undefined8 in_d24;
  undefined8 in_register_00005308;
  undefined8 in_d25;
  undefined8 in_register_00005328;
  undefined8 in_d26;
  undefined8 in_register_00005348;
  undefined8 in_d27;
  undefined8 in_register_00005368;
  undefined8 in_d28;
  undefined8 in_register_00005388;
  undefined8 in_d29;
  undefined8 in_register_000053a8;
  undefined8 in_d30;
  undefined8 in_register_000053c8;
  undefined8 in_d31;
```

```
  undefined8 in_register_000053e8;
  ulong hcr_el2;
  ulong ESR_GL1;
  ulong ELR_GL1;
  ulong FAR_GL1;
  ulong ESR_GL1__;
  TPIDR *tmp_TPIDR;
  TPIDR *TPIDR;
  ulong ?daif?;
  ulong ??daif?;
  ulong SPSR_GL1;
  undefined8 SPSR_GL1_store;
  long TPIDR_GL2;

  pan = 0;
  tmp_TPIDR = (TPIDR *)UnkSytemRegRead(3,6,0xf,0xb,1);
  do {
  } while (&stack0x00000000 != *(undefined1
   ↪  **)&tmp_TPIDR->field_0x470);
  hcr_el2 = hcr_el2;
  if ((hcr_el2 & 0x8000001) == 1) {
    pan = 0;
    TPIDR = (TPIDR *)UnkSytemRegRead(3,6,0xf,0xb,1);
    *(undefined8 *)&TPIDR->field_0x650 = param_12;
    *(undefined8 *)&TPIDR->field_0x658 = param_13;
    *(undefined8 *)&TPIDR->field_0x660 = param_14;
    *(undefined8 *)&TPIDR->field_0x668 = param_15;
    *(undefined8 *)&TPIDR->field_0x670 = param_16;
    *(undefined8 *)&TPIDR->field_0x678 = param_17;
    *(undefined8 *)&TPIDR->field_0x680 = param_1;
    *(undefined8 *)&TPIDR->field_0x688 = in_x9;
    *(undefined8 *)&TPIDR->field_0x690 = in_x10;
    *(undefined8 *)&TPIDR->field_0x698 = in_x11;
    *(undefined8 *)&TPIDR->field_0x6a0 = in_x12;
    *(undefined8 *)&TPIDR->field_0x6a8 = in_x13;
    *(undefined8 *)&TPIDR->field_0x6b0 = in_x14;
    *(undefined8 *)&TPIDR->field_0x6b8 = in_x15;
    *(ulong *)&TPIDR->field_0x6c0 = in_x16;
    *(undefined8 *)&TPIDR->field_0x6c8 = in_x17;
    *(undefined8 *)&TPIDR->field_0x6d0 = in_x18;
    *(undefined8 *)&TPIDR->field_0x6d8 = unaff_x19;
    *(undefined8 *)&TPIDR->field_0x6e0 = unaff_x20;
    *(undefined8 *)&TPIDR->field_0x6e8 = unaff_x21;
    *(undefined8 *)&TPIDR->field_0x6f0 = unaff_x22;
    *(undefined8 *)&TPIDR->field_0x6f8 = unaff_x23;
    *(undefined8 *)&TPIDR->field_0x700 = unaff_x24;
    *(undefined8 *)&TPIDR->field_0x708 = unaff_x25;
    *(undefined8 *)&TPIDR->field_0x710 = unaff_x26;
    *(undefined8 *)&TPIDR->field_0x718 = unaff_x27;
    *(undefined8 *)&TPIDR->field_0x720 = unaff_x28;
    *(undefined8 *)&TPIDR->field_0x728 = unaff_x29;
    *(undefined8 *)&TPIDR->field_0x730 = unaff_x30;
    *(long *)&TPIDR->field_0x768 = param_2._8_8_;
    *(long *)&TPIDR->field_0x760 = param_2._0_8_;
    *(long *)&TPIDR->field_0x778 = param_3._8_8_;
    *(long *)&TPIDR->field_0x770 = param_3._0_8_;
    *(long *)&TPIDR->field_0x788 = param_4._8_8_;
```

```
145     *(long *)&TPIDR->field_0x780 = param_4._0_8_;
146     *(long *)&TPIDR->field_0x798 = param_5._8_8_;
147     *(long *)&TPIDR->field_0x790 = param_5._0_8_;
148     *(long *)&TPIDR->field_0x7a8 = param_6._8_8_;
149     *(long *)&TPIDR->field_0x7a0 = param_6._0_8_;
150     *(long *)&TPIDR->field_0x7b8 = param_7._8_8_;
151     *(long *)&TPIDR->field_0x7b0 = param_7._0_8_;
152     *(long *)&TPIDR->field_0x7c8 = param_8._8_8_;
153     *(long *)&TPIDR->field_0x7c0 = param_8._0_8_;
154     *(long *)&TPIDR->field_0x7d8 = param_9._8_8_;
155     *(long *)&TPIDR->field_0x7d0 = param_9._0_8_;
156     *(undefined8 *)&TPIDR->field_0x7e8 = in_register_00005108;
157     *(undefined8 *)&TPIDR->field_0x7e0 = unaff_d8;
158     *(undefined8 *)&TPIDR->field_0x7f8 = in_register_00005128;
159     *(undefined8 *)&TPIDR->field_0x7f0 = unaff_d9;
160     *(undefined8 *)&TPIDR->field_0x808 = in_register_00005148;
161     *(undefined8 *)&TPIDR->field_0x800 = unaff_d10;
162     *(undefined8 *)&TPIDR->field_0x818 = in_register_00005168;
163     *(undefined8 *)&TPIDR->field_0x810 = unaff_d11;
164     *(undefined8 *)&TPIDR->field_0x828 = in_register_00005188;
165     *(undefined8 *)&TPIDR->field_0x820 = unaff_d12;
166     *(undefined8 *)&TPIDR->field_0x838 = in_register_000051a8;
167     *(undefined8 *)&TPIDR->field_0x830 = unaff_d13;
168     *(undefined8 *)&TPIDR->field_0x848 = in_register_000051c8;
169     *(undefined8 *)&TPIDR->field_0x840 = unaff_d14;
170     *(undefined8 *)&TPIDR->field_0x858 = in_register_000051e8;
171     *(undefined8 *)&TPIDR->field_0x850 = unaff_d15;
172     *(undefined8 *)&TPIDR->field_0x868 = in_register_00005208;
173     *(undefined8 *)&TPIDR->field_0x860 = in_d16;
174     *(undefined8 *)&TPIDR->field_0x878 = in_register_00005228;
175     *(undefined8 *)&TPIDR->field_0x870 = in_d17;
176     *(undefined8 *)&TPIDR->field_0x888 = in_register_00005248;
177     *(undefined8 *)&TPIDR->field_0x880 = in_d18;
178     *(undefined8 *)&TPIDR->field_0x898 = in_register_00005268;
179     *(undefined8 *)&TPIDR->field_0x890 = in_d19;
180     *(undefined8 *)&TPIDR->field_0x8a8 = in_register_00005288;
181     *(undefined8 *)&TPIDR->field_0x8a0 = in_d20;
182     *(undefined8 *)&TPIDR->field_0x8b8 = in_register_000052a8;
183     *(undefined8 *)&TPIDR->field_0x8b0 = in_d21;
184     *(undefined8 *)&TPIDR->field_0x8c8 = in_register_000052c8;
185     *(undefined8 *)&TPIDR->field_0x8c0 = in_d22;
186     *(undefined8 *)&TPIDR->field_0x8d8 = in_register_000052e8;
187     *(undefined8 *)&TPIDR->field_0x8d0 = in_d23;
188     *(undefined8 *)&TPIDR->field_0x8e8 = in_register_00005308;
189     *(undefined8 *)&TPIDR->field_0x8e0 = in_d24;
190     *(undefined8 *)&TPIDR->field_0x8f8 = in_register_00005328;
191     *(undefined8 *)&TPIDR->field_0x8f0 = in_d25;
192     *(undefined8 *)&TPIDR->field_0x908 = in_register_00005348;
193     *(undefined8 *)&TPIDR->field_0x900 = in_d26;
194     *(undefined8 *)&TPIDR->field_0x918 = in_register_00005368;
195     *(undefined8 *)&TPIDR->field_0x910 = in_d27;
196     *(undefined8 *)&TPIDR->field_0x928 = in_register_00005388;
197     *(undefined8 *)&TPIDR->field_0x920 = in_d28;
198     *(undefined8 *)&TPIDR->field_0x938 = in_register_000053a8;
199     *(undefined8 *)&TPIDR->field_0x930 = in_d29;
200     *(undefined8 *)&TPIDR->field_0x948 = in_register_000053c8;
201     *(undefined8 *)&TPIDR->field_0x940 = in_d30;
```

```
202     *(undefined8 *)&TPIDR->field_0x950 = in_d31;
203     *(undefined8 *)&TPIDR->field_0x958 = in_register_000053e8;
204     ELR_GL1 = UnkSytemRegRead(3,6,0xf,10,6);
205     SPSR_GL1_store = UnkSytemRegRead(3,6,0xf,10,3);
206     uVar2 = fpsr;
207     uVar1 = fpcr;
208     *(ulong *)&TPIDR->ELR_store = ELR_GL1;
209     *(int *)&TPIDR->SPSR_store = (int)SPSR_GL1_store;
210     *(int *)&TPIDR->floating_point_status_register = (int)uVar2;
211     *(int *)&TPIDR->floating_point_control_register = (int)uVar1;
212     FAR_GL1 = UnkSytemRegRead(3,6,0xf,10,7);
213     ESR_GL1__ = UnkSytemRegRead(3,6,0xf,10,5);
214     *(ulong *)&TPIDR->FAR_store = FAR_GL1;
215     *(ulong *)&TPIDR->ESR_store = ESR_GL1__;
216     *(undefined8 *)&TPIDR->field_0x640 = param_10;
217     *(ulong *)&TPIDR->field_0x648 = param_11;
218     SPSR_GL1_store = sp_el0;
219     *(undefined8 *)&TPIDR->field_0x738 = SPSR_GL1_store;
220     calls_CSPTM_event6_x16_param2(3,3);
221     return;
222   }
223                     /* GL1 Exception Syndrome Register
224                        */
225   ESR_GL1 = UnkSytemRegRead(3,6,0xf,10,5);
226   EC = ESR_GL1 >> 0x1a & 0x3f;
227   if (EC == 0b00010101) {
228     SPSR_GL1 = UnkSytemRegRead(3,6,0xf,10,5);
229     SVC_Imm = SPSR_GL1 & 0xffff;
230                     /* we have SVC 0 */
231     if (SVC_Imm == 0) {
232       dispatch_table = in_x16 >> 0x20 & 0xff;
233       if (dispatch_table == 0xfd) {
234         SVC_0_is_0xfd/RETURN_TO_CALLER();
235         return;
236       }
237       if (dispatch_table != 0xfe) {
238         if (dispatch_table != 0xff) {
239           TPIDR_GL2 = UnkSytemRegRead(3,6,0xf,0xb,1);
240           lVar5 = *(long *)(TPIDR_GL2 + 0x470);
241           *(undefined8 *)(TPIDR_GL2 + 0x490) = param_10;
242           *(ulong *)(TPIDR_GL2 + 0x498) = param_11;
243           *(undefined8 *)(TPIDR_GL2 + 0x4a0) = param_12;
244           *(undefined8 *)(TPIDR_GL2 + 0x4a8) = param_13;
245           *(undefined8 *)(TPIDR_GL2 + 0x4b0) = param_14;
246           *(undefined8 *)(TPIDR_GL2 + 0x4b8) = param_15;
247           *(undefined8 *)(TPIDR_GL2 + 0x4c0) = param_16;
248           *(undefined8 *)(TPIDR_GL2 + 0x4c8) = param_17;
249           some_flag_from_TPIDR_GL2 = *(ulong *)(TPIDR_GL2 + 0x488);
250           do {
251           } while (1 < some_flag_from_TPIDR_GL2);
252           do {
253           } while (1 < some_flag_from_TPIDR_GL2);
254           if (some_flag_from_TPIDR_GL2 == 1) {
255             puVar6 = (undefined8 *)(TPIDR_GL2 + 0x588);
256           }
257           else {
258             puVar6 = (undefined8 *)(TPIDR_GL2 + 0x4d0);
```

```
259             }
260             *puVar6 = unaff_x19;
261             puVar6[1] = unaff_x20;
262                       /* we basically put TPIDR_GL2 on the stack? */
263             puVar6[2] = unaff_x21;
264             puVar6[3] = unaff_x22;
265             puVar6[4] = unaff_x23;
266             puVar6[5] = unaff_x24;
267             puVar6[6] = unaff_x25;
268             puVar6[7] = unaff_x26;
269             puVar6[8] = unaff_x27;
270             puVar6[9] = unaff_x28;
271             puVar6[0x12] = unaff_x29;
272             puVar6[0x13] = unaff_x30;
273             puVar6[10] = unaff_d8;
274             puVar6[0xb] = unaff_d9;
275             puVar6[0xc] = unaff_d10;
276             puVar6[0xd] = unaff_d11;
277             puVar6[0xe] = unaff_d12;
278             puVar6[0xf] = unaff_d13;
279             puVar6[0x10] = unaff_d14;
280             puVar6[0x11] = unaff_d15;
281             *(ulong *)(TPIDR_GL2 + 0x488) = some_flag_from_TPIDR_GL2 + 1;
282                       /* SPSR_GL1 - holds the saved process stat when an
                           ↪  exception is taken to EL1 */
283             SPSR_GL1_store = UnkSytemRegRead(3,6,0xf,10,3);
284             puVar6[0x16] = SPSR_GL1_store;
285             SPSR_GL1_store = sp_el0;
286             puVar6[0x14] = SPSR_GL1_store;
287                       /* ELR_GL1 - when taking an exception to EL1, holds
                           ↪  the address to return to
288                          */
289             SPSR_GL1_store = UnkSytemRegRead(3,6,0xf,10,6);
290             puVar6[0x15] = SPSR_GL1_store;
291             *(undefined8 *)(lVar5 + -0x10) = unaff_x29;
292             *(undefined8 *)(lVar5 + -8) = unaff_x30;
293             CORE_SPTM_FUNCTION(2);
294             return;
295           }
296           SVC_0_is_0xff();
297           return;
298         }
299                       /* TPIDR_GL0 */
300         TPIDR_GL2 = UnkSytemRegRead(3,6,0xf,0xb,1);
301                       /* SPRR_UPERM_EL0 - SPRR User Permissions
                           ↪  Configuration Register
302                          we set this to a value from TPIDR_GL2 */
303         UnkSytemRegWrite(3,6,0xf,1,5,*(undefined8 *)(TPIDR_GL2 + 0x9a0));
304         mdscr_el1 = *(undefined8 *)(TPIDR_GL2 + 0x9a8);
305                       /* AGTCNTRDIR_EL1 */
306         UnkSytemRegWrite(3,1,0xf,1,5,*(undefined8 *)(TPIDR_GL2 + 0x9b0));
307                       /* 3_4_c15_c0_4 */
308         UnkSytemRegWrite(3,4,0xf,0,4,*(undefined8 *)(TPIDR_GL2 + 0x9b8));
309         sctlr_el1 = *(undefined8 *)(TPIDR_GL2 + 0x9c0);
310         tcr_el1 = *(undefined8 *)(TPIDR_GL2 + 0x9c8);
311         spsel = 0;
312         calls_CSPTM_both_variable(param_10,0,1);
```

```
      return;
    }
    if (SVC_Imm == 0x25) {
      ?daif? = UnkSytemRegRead(3,6,0xf,10,3);
      UnkSytemRegWrite(3,6,0xf,10,3,?daif? & 0xffffffffffffe3f);
      ExceptionReturn();
      return;
    }
    if (SVC_Imm == 0x26) {
      ??daif? = UnkSytemRegRead(3,6,0xf,10,3);
      UnkSytemRegWrite(3,6,0xf,10,3,??daif? | 0x1c0);
      ExceptionReturn();
      return;
    }
    do {
      WaitForEvent();
    } while( true );
  }
  if (EC != 0b00010110) {
    TPIDR_GL2 = UnkSytemRegRead(3,6,0xf,0xb,1);
    *(undefined8 *)(TPIDR_GL2 + 0x640) = param_10;
    *(ulong *)(TPIDR_GL2 + 0x648) = param_11;
    SPSR_GL1_store = sp_el0;
    *(undefined8 *)(TPIDR_GL2 + 0x738) = SPSR_GL1_store;
    *(undefined8 *)(TPIDR_GL2 + 0x650) = param_12;
    *(undefined8 *)(TPIDR_GL2 + 0x658) = param_13;
    *(undefined8 *)(TPIDR_GL2 + 0x660) = param_14;
    *(undefined8 *)(TPIDR_GL2 + 0x668) = param_15;
    *(undefined8 *)(TPIDR_GL2 + 0x670) = param_16;
    *(undefined8 *)(TPIDR_GL2 + 0x678) = param_17;
    *(undefined8 *)(TPIDR_GL2 + 0x680) = param_1;
    *(undefined8 *)(TPIDR_GL2 + 0x688) = in_x9;
    *(undefined8 *)(TPIDR_GL2 + 0x690) = in_x10;
    *(undefined8 *)(TPIDR_GL2 + 0x698) = in_x11;
    *(undefined8 *)(TPIDR_GL2 + 0x6a0) = in_x12;
    *(undefined8 *)(TPIDR_GL2 + 0x6a8) = in_x13;
    *(undefined8 *)(TPIDR_GL2 + 0x6b0) = in_x14;
    *(undefined8 *)(TPIDR_GL2 + 0x6b8) = in_x15;
    *(ulong *)(TPIDR_GL2 + 0x6c0) = in_x16;
    *(undefined8 *)(TPIDR_GL2 + 0x6c8) = in_x17;
    *(undefined8 *)(TPIDR_GL2 + 0x6d0) = in_x18;
    *(undefined8 *)(TPIDR_GL2 + 0x6d8) = unaff_x19;
    *(undefined8 *)(TPIDR_GL2 + 0x6e0) = unaff_x20;
    *(undefined8 *)(TPIDR_GL2 + 0x6e8) = unaff_x21;
    *(undefined8 *)(TPIDR_GL2 + 0x6f0) = unaff_x22;
    *(undefined8 *)(TPIDR_GL2 + 0x6f8) = unaff_x23;
    *(undefined8 *)(TPIDR_GL2 + 0x700) = unaff_x24;
    *(undefined8 *)(TPIDR_GL2 + 0x708) = unaff_x25;
    *(undefined8 *)(TPIDR_GL2 + 0x710) = unaff_x26;
    *(undefined8 *)(TPIDR_GL2 + 0x718) = unaff_x27;
    *(undefined8 *)(TPIDR_GL2 + 0x720) = unaff_x28;
    *(undefined8 *)(TPIDR_GL2 + 0x728) = unaff_x29;
    *(undefined8 *)(TPIDR_GL2 + 0x730) = unaff_x30;
    *(long *)(TPIDR_GL2 + 0x768) = param_2._8_8_;
    *(long *)(TPIDR_GL2 + 0x760) = param_2._0_8_;
    *(long *)(TPIDR_GL2 + 0x778) = param_3._8_8_;
    *(long *)(TPIDR_GL2 + 0x770) = param_3._0_8_;
```

```
    *(long *)(TPIDR_GL2 + 0x788) = param_4._8_8_;
    *(long *)(TPIDR_GL2 + 0x780) = param_4._0_8_;
    *(long *)(TPIDR_GL2 + 0x798) = param_5._8_8_;
    *(long *)(TPIDR_GL2 + 0x790) = param_5._0_8_;
    *(long *)(TPIDR_GL2 + 0x7a8) = param_6._8_8_;
    *(long *)(TPIDR_GL2 + 0x7a0) = param_6._0_8_;
    *(long *)(TPIDR_GL2 + 0x7b8) = param_7._8_8_;
    *(long *)(TPIDR_GL2 + 0x7b0) = param_7._0_8_;
    *(long *)(TPIDR_GL2 + 0x7c8) = param_8._8_8_;
    *(long *)(TPIDR_GL2 + 0x7c0) = param_8._0_8_;
    *(long *)(TPIDR_GL2 + 0x7d8) = param_9._8_8_;
    *(long *)(TPIDR_GL2 + 2000) = param_9._0_8_;
    *(undefined8 *)(TPIDR_GL2 + 0x7e8) = in_register_00005108;
    *(undefined8 *)(TPIDR_GL2 + 0x7e0) = unaff_d8;
    *(undefined8 *)(TPIDR_GL2 + 0x7f8) = in_register_00005128;
    *(undefined8 *)(TPIDR_GL2 + 0x7f0) = unaff_d9;
    *(undefined8 *)(TPIDR_GL2 + 0x808) = in_register_00005148;
    *(undefined8 *)(TPIDR_GL2 + 0x800) = unaff_d10;
    *(undefined8 *)(TPIDR_GL2 + 0x818) = in_register_00005168;
    *(undefined8 *)(TPIDR_GL2 + 0x810) = unaff_d11;
    *(undefined8 *)(TPIDR_GL2 + 0x828) = in_register_00005188;
    *(undefined8 *)(TPIDR_GL2 + 0x820) = unaff_d12;
    *(undefined8 *)(TPIDR_GL2 + 0x838) = in_register_000051a8;
    *(undefined8 *)(TPIDR_GL2 + 0x830) = unaff_d13;
    *(undefined8 *)(TPIDR_GL2 + 0x848) = in_register_000051c8;
    *(undefined8 *)(TPIDR_GL2 + 0x840) = unaff_d14;
    *(undefined8 *)(TPIDR_GL2 + 0x858) = in_register_000051e8;
    *(undefined8 *)(TPIDR_GL2 + 0x850) = unaff_d15;
    *(undefined8 *)(TPIDR_GL2 + 0x868) = in_register_00005208;
    *(undefined8 *)(TPIDR_GL2 + 0x860) = in_d16;
    *(undefined8 *)(TPIDR_GL2 + 0x878) = in_register_00005228;
    *(undefined8 *)(TPIDR_GL2 + 0x870) = in_d17;
    *(undefined8 *)(TPIDR_GL2 + 0x888) = in_register_00005248;
    *(undefined8 *)(TPIDR_GL2 + 0x880) = in_d18;
    *(undefined8 *)(TPIDR_GL2 + 0x898) = in_register_00005268;
    *(undefined8 *)(TPIDR_GL2 + 0x890) = in_d19;
    *(undefined8 *)(TPIDR_GL2 + 0x8a8) = in_register_00005288;
    *(undefined8 *)(TPIDR_GL2 + 0x8a0) = in_d20;
    *(undefined8 *)(TPIDR_GL2 + 0x8b8) = in_register_000052a8;
    *(undefined8 *)(TPIDR_GL2 + 0x8b0) = in_d21;
    *(undefined8 *)(TPIDR_GL2 + 0x8c8) = in_register_000052c8;
    *(undefined8 *)(TPIDR_GL2 + 0x8c0) = in_d22;
    *(undefined8 *)(TPIDR_GL2 + 0x8d8) = in_register_000052e8;
    *(undefined8 *)(TPIDR_GL2 + 0x8d0) = in_d23;
    *(undefined8 *)(TPIDR_GL2 + 0x8e8) = in_register_00005308;
    *(undefined8 *)(TPIDR_GL2 + 0x8e0) = in_d24;
    *(undefined8 *)(TPIDR_GL2 + 0x8f8) = in_register_00005328;
    *(undefined8 *)(TPIDR_GL2 + 0x8f0) = in_d25;
    *(undefined8 *)(TPIDR_GL2 + 0x908) = in_register_00005348;
    *(undefined8 *)(TPIDR_GL2 + 0x900) = in_d26;
    *(undefined8 *)(TPIDR_GL2 + 0x918) = in_register_00005368;
    *(undefined8 *)(TPIDR_GL2 + 0x910) = in_d27;
    *(undefined8 *)(TPIDR_GL2 + 0x928) = in_register_00005388;
    *(undefined8 *)(TPIDR_GL2 + 0x920) = in_d28;
    *(undefined8 *)(TPIDR_GL2 + 0x938) = in_register_000053a8;
    *(undefined8 *)(TPIDR_GL2 + 0x930) = in_d29;
    *(undefined8 *)(TPIDR_GL2 + 0x948) = in_register_000053c8;
```

```
    *(undefined8 *)(TPIDR_GL2 + 0x940) = in_d30;
    *(undefined8 *)(TPIDR_GL2 + 0x950) = in_d31;
    *(undefined8 *)(TPIDR_GL2 + 0x958) = in_register_000053e8;
    SPSR_GL1_store = UnkSytemRegRead(3,6,0xf,10,6);
    uVar1 = UnkSytemRegRead(3,6,0xf,10,3);
    uVar3 = fpsr;
    uVar2 = fpcr;
    *(undefined8 *)(TPIDR_GL2 + 0x740) = SPSR_GL1_store;
    *(int *)(TPIDR_GL2 + 0x748) = (int)uVar1;
    *(int *)(TPIDR_GL2 + 0x960) = (int)uVar3;
    *(int *)(TPIDR_GL2 + 0x964) = (int)uVar2;
    SPSR_GL1_store = UnkSytemRegRead(3,6,0xf,10,7);
    uVar1 = UnkSytemRegRead(3,6,0xf,10,5);
    *(undefined8 *)(TPIDR_GL2 + 0x750) = SPSR_GL1_store;
    *(undefined8 *)(TPIDR_GL2 + 0x758) = uVar1;
    lVar5 = UnkSytemRegRead(3,6,0xf,0xb,1);
    lVar5 = *(long *)(lVar5 + 0x470);
    SPSR_GL1_store = UnkSytemRegRead(3,6,0xf,10,6);
    *(undefined8 *)(lVar5 + -0x10) = unaff_x29;
    *(undefined8 *)(lVar5 + -8) = SPSR_GL1_store;
    lVar5 = hcr_el2;
    if (lVar5 == 0x30480000000) {
      SPSR_GL1_store = 2;
      SPSR_GL1 = UnkSytemRegRead(3,6,0xf,10,3);
      if ((SPSR_GL1 & 0xc) == 0) {
        pcVar4 = "[SK] Unhandled synchronous exception taken from GL0";
      }
      else {
        pcVar4 = "[SK] Unhandled synchronous exception taken from GL1";
      }
    }
    else {
      SPSR_GL1_store = 1;
      pcVar4 = "[TXM] Unhandled synchronous exception taken from GL0";
    }
    if (DAT_fffffff027085028 != 0) {
      *(undefined8 *)(DAT_fffffff027085028 + 0x40) = SPSR_GL1_store;
    }
    FUN_fffffff0270cc638((undefined8 *)(TPIDR_GL2 + 0x640),pcVar4);
    return;
  }
  dispatch_table_id = in_x16 >> 0x20 & 0xff;
  if (dispatch_table_id == 0xfd) {
    FUN_fffffff027082530();
    return;
  }
  if (dispatch_table_id == 0xfe) {
    FUN_fffffff027082218();
    return;
  }
  if (dispatch_table_id == 0xff) {
    TPIDR_GL2 = UnkSytemRegRead(3,6,0xf,0xb,1);
    lVar5 = *(long *)(TPIDR_GL2 + 0x470);
    *(undefined8 *)(lVar5 + -0x10) = 0;
    *(undefined1 **)(lVar5 + -8) = &LAB_fffffff0270825b0;
    *(undefined8 *)(TPIDR_GL2 + 0x998) = param_10;
    SPSR_GL1 = *(long *)(TPIDR_GL2 + 0x488) - 1;
```

```
484     do {
485     } while (1 < SPSR_GL1);
486     do {
487     } while (1 < SPSR_GL1);
488     if (SPSR_GL1 == 1) {
489       lVar5 = TPIDR_GL2 + 0x588;
490     }
491     else {
492       lVar5 = TPIDR_GL2 + 0x4d0;
493     }
494     SPSR_GL1_store = *(undefined8 *)(lVar5 + 0x90);
495     uVar1 = *(undefined8 *)(lVar5 + 0x98);
496     *(undefined1 **)(TPIDR_GL2 + 0x740) = &LAB_fffffff0270825b0;
497     *(undefined8 *)(TPIDR_GL2 + 0x728) = SPSR_GL1_store;
498     *(undefined8 *)(TPIDR_GL2 + 0x730) = uVar1;
499     CORE_SPTM_FUNCTION(7,(param_11 & 1) << 2 | 2);
500     return;
501   }
502   leads_to_CORE_SPTM_FUNCTION();
503   return;
504 }
505
```

Listing A.3: `synchronous_handler_from_lower` function in SPTM responsible for handling incoming SVC and HVC exceptions.

A.10 SPTM SVC Handling Excerpt from `synchronous_exception_handler_from_lower`

```
1                               SVC_HANDLER
2       ff027081f88 a8 fa 3e d5     mrs         EC,sreg(0x3, 0x6, c0xf,
         ↪  c0xa, 0x5)
3       ff027081f8c 08 3d 40 92     and         SVC_Imm,SVC_Imm,#0xffff
4                               we have SVC 0
5       ff027081f90 1f 01 00 f1     cmp         SVC_Imm,#0x0
6       ff027081f94 e0 02 00 54     b.eq        SVC_0_HANDLER
7       ff027081f98 1f 95 00 f1     cmp         SVC_Imm,#0x25
8       ff027081f9c c0 00 00 54     b.eq
         ↪  SVC_37_HANDLER_enableInterrupts
9       ff027081fa0 1f 99 00 f1     cmp         SVC_Imm,#0x26
10      ff027081fa4 80 01 00 54     b.eq
         ↪  SVC_38_HANDLER_disableInterrupts
11      ff027081fa8 a0 d5 9b d2     mov         x0,#0xdead
12                              LAB_fffffff027081fac
13      ff027081fac 5f 20 03 d5     wfe
14      ff027081fb0 ff ff ff 17     b           LAB_fffffff027081fac
15
```

Listing A.4: SPTM synchronous_exception_handler_from_lower function.

A.11 SPTM Frame Types

Type	Value
SPTM_UNTYPED	0
SPTM_UNUSED	1
SPTM_DEFAULT	2
SPTM_RO	3
SPTM_CODE	4
SPTM_TXM_CODE	5
SPTM_XNU_CODE	6
SPTM_XNU_CODE_DBG_RW	7
SPTM_KERNEL_ROOT_TABLE	8
SPTM_PAGE_TABLE	9
SPTM_IOMMU_BOOTSTRAP	10
XNU_DEFAULT	11
XNU_RO	12
XNU_RO_DBG_RW	13
XNU_USER_EXEC	14
XNU_USER_DEBUG	15
XNU_USER_JIT	16
XNU_USER_ROOT_TABLE	17
XNU_SHARED_ROOT_TABLE	18
XNU_PAGE_TABLE	19
XNU_PAGE_TABLE_SHARED	20
XNU_PAGE_TABLE_ROZONE	21
XNU_PAGE_TABLE_COMMPAGE	22
XNU_IOMMU	23
XNU_ROZONE	24
XNU_IO	25
XNU_PROTECTED_IO	26
XNU_COMMPAGE_RW	27
XNU_COMMPAGE_RO	28
XNU_COMMPAGE_RX	29
XNU_TAG_STORAGE	30
XNU_STAGE2_ROOT_TABLE	31
XNU_STAGE2_PAGE_TABLE	32
XNU_KERNEL_RESTRICTED	33
XNU_RESERVED_1	34
XNU_RESERVED_2	35
XNU_RESTRICTED_IO	36
XNU_RESTRICTED_IO_TELEMETRY	37
TXM_DEFAULT	38
TXM_RO	39
TXM_RW	40
TXM_CPU_STACK	41
TXM_THREAD_STACK	42
TXM_ADDRESS_SPACE_TABLE	43
TXM_MALLOC_PAGE	44
TXM_FREE_LIST	45
TXM_SLAB_TRUST_CACHE	46
TXM_SLAB_PROFILE	47
TXM_SLAB_CODE_SIGNATURE	48
TXM_SLAB_CODE_REGION	49
TXM_SLAB_ADDRESS_SPACE	50
TXM_BUCKET_1024	51
TXM_BUCKET_2048	52
TXM_BUCKET_4096	53
TXM_BUCKET_8192	54
TXM_BULK_DATA	55
TXM_BULK_DATA_READ_ONLY	56
TXM_LOG	57

Type	Value
TXM_SEP_SECURE_CHANNEL	58
SK_DEFAULT	59
SK_SHARED_RO	60
SK_SHARED_RW	61
SK_IO	62

Table A.7: SPTM frame types and their corresponding integer values [3].

A.12 Retyping Allowed Caller Domains

Type	Type Name	Domain	Domain Name	Retype Flag
0	SPTM_UNTYPED	0	SPTM_DOMAIN	0
1	SPTM_UNUSED	0	SPTM_DOMAIN	0
2	SPTM_DEFAULT	0	SPTM_DOMAIN	0
3	SPTM_RO	0	SPTM_DOMAIN	0
4	SPTM_CODE	0	SPTM_DOMAIN	0
5	SPTM_TXM_CODE	0	SPTM_DOMAIN	0
6	SPTM_XNU_CODE	0	SPTM_DOMAIN	0
7	SPTM_XNU_CODE_DBG_RW	0	SPTM_DOMAIN	0
8	SPTM_KERNEL_ROOT_TABLE	0	SPTM_DOMAIN	0
9	SPTM_PAGE_TABLE	0	SPTM_DOMAIN	255
10	SPTM_IOMMU_BOOTSTRAP	0	SPTM_DOMAIN	0
11	XNU_DEFAULT	1	XNU_DOMAIN	3840
12	XNU_RO	1	XNU_DOMAIN	3840
13	XNU_RO_DBG_RW	1	XNU_DOMAIN	3840
14	XNU_USER_EXEC	1	XNU_DOMAIN	3840
15	XNU_USER_DEBUG	1	XNU_DOMAIN	3840
16	XNU_USER_JIT	1	XNU_DOMAIN	3840
17	XNU_USER_ROOT_TABLE	1	XNU_DOMAIN	48
18	XNU_SHARED_ROOT_TABLE	1	XNU_DOMAIN	0
19	XNU_PAGE_TABLE	1	XNU_DOMAIN	0
20	XNU_PAGE_TABLE_SHARED	1	XNU_DOMAIN	0
21	XNU_PAGE_TABLE_ROZONE	1	XNU_DOMAIN	0
22	XNU_PAGE_TABLE_COMMPAGE	1	XNU_DOMAIN	0
23	XNU_IOMMU	1	XNU_DOMAIN	0
24	XNU_ROZONE	1	XNU_DOMAIN	3840
25	XNU_IO	1	XNU_DOMAIN	0
26	XNU_PROTECTED_IO	1	XNU_DOMAIN	0
27	XNU_COMMPAGE_RW	1	XNU_DOMAIN	0
28	XNU_COMMPAGE_RO	1	XNU_DOMAIN	0
29	XNU_COMMPAGE_RX	1	XNU_DOMAIN	0
30	XNU_TAG_STORAGE	0	SPTM_DOMAIN	0
31	XNU_STAGE2_ROOT_TABLE	1	XNU_DOMAIN	0
32	XNU_STAGE2_PAGE_TABLE	1	XNU_DOMAIN	0
33	XNU_KERNEL_RESTRICTED	1	XNU_DOMAIN	3840
34	XNU_RESERVED_1	1	XNU_DOMAIN	0
35	XNU_RESERVED_2	1	XNU_DOMAIN	0
36	XNU_RESTRICTED_IO	1	XNU_DOMAIN	0
37	XNU_RESTRICTED_IO_TELEMETRY	1	XNU_DOMAIN	0
38	TXM_DEFAULT	2	TXM_DOMAIN	3840
39	TXM_RO	2	TXM_DOMAIN	0
40	TXM_RW	2	TXM_DOMAIN	0

Type	Type Name	Domain	Domain Name	Retype Flag
41	TXM_CPU_STACK	2	TXM_DOMAIN	0
42	TXM_THREAD_STACK	2	TXM_DOMAIN	0
43	TXM_ADDRESS_SPACE_TABLE	2	TXM_DOMAIN	0
44	TXM_MALLOC_PAGE	2	TXM_DOMAIN	0
45	TXM_FREE_LIST	2	TXM_DOMAIN	3840
46	TXM_SLAB_TRUST_CACHE	2	TXM_DOMAIN	0
47	TXM_SLAB_PROFILE	2	TXM_DOMAIN	0
48	TXM_SLAB_CODE_SIGNATURE	2	TXM_DOMAIN	0
49	TXM_SLAB_CODE_REGION	2	TXM_DOMAIN	0
50	TXM_SLAB_ADDRESS_SPACE	2	TXM_DOMAIN	0
51	TXM_BUCKET_1024	2	TXM_DOMAIN	0
52	TXM_BUCKET_2048	2	TXM_DOMAIN	0
53	TXM_BUCKET_4096	2	TXM_DOMAIN	0
54	TXM_BUCKET_8192	2	TXM_DOMAIN	0
55	TXM_BULK_DATA	2	TXM_DOMAIN	3840
56	TXM_BULK_DATA_READ_ONLY	2	TXM_DOMAIN	3840
57	TXM_LOG	2	TXM_DOMAIN	0
58	TXM_SEP_SECURE_CHANNEL	2	TXM_DOMAIN	0
59	SK_DEFAULT	3	SK_DOMAIN	0
60	SK_SHARED_RO	3	SK_DOMAIN	3840
61	SK_SHARED_RW	3	SK_DOMAIN	3840
62	SK_IO	3	SK_DOMAIN	0

Table A.8: The table shows valid caller domains for memory frame retyping of frames of the specified type via the SPTM `retype` function. The domain ID is resolved to the corresponding domain. The table also contains the retyping flag that is used for conditional handling in the process.

A.13 SPTM Type to SPRR Index Mapping

Index	Name	Value
0	SPTM_UNTYPED	0x1
1	SPTM_UNUSED	0xff
2	SPTM_DEFAULT	0x1
3	SPTM_RO	0xb
4	SPTM_CODE	0x0
5	SPTM_TXM_CODE	0x8
6	SPTM_XNU_CODE	0xa
7	SPTM_XNU_CODE_DBG_RW	0xa
8	SPTM_KERNEL_ROOT_TABLE	0x1
9	SPTM_PAGE_TABLE	0x1
10	SPTM_IOMMU_BOOTSTRAP	0x1
11	XNU_DEFAULT	0x3
12	XNU_RO	0xb
13	XNU_RO_DBG_RW	0xb
14	XNU_USER_EXEC	0xb
15	XNU_USER_DEBUG	0xb
16	XNU_USER_JIT	0xb
17	XNU_USER_ROOT_TABLE	0x1
18	XNU_SHARED_ROOT_TABLE	0x1

Index	Name	Value
19	XNU_PAGE_TABLE	0x1
20	XNU_PAGE_TABLE_SHARED	0x1
21	XNU_PAGE_TABLE_ROZONE	0x1
22	XNU_PAGE_TABLE_COMMPAGE	0x1
23	XNU_IOMMU	0x1
24	XNU_ROZONE	0x2
25	XNU_IO	0xff
26	XNU_PROTECTED_IO	0x1
27	XNU_COMMPAGE_RW	0x3
28	XNU_COMMPAGE_RO	0xb
29	XNU_COMMPAGE_RX	0xb
30	XNU_TAG_STORAGE	0x0
31	XNU_STAGE2_ROOT_TABLE	0x1
32	XNU_STAGE2_PAGE_TABLE	0x1
33	XNU_KERNEL_RESTRICTED	0x3
34	XNU_RESERVED_1	0xb
35	XNU_RESERVED_2	0xb
36	XNU_RESTRICTED_IO	0xff
37	XNU_RESTRICTED_IO_TELEMETRY	0xff
38	TXM_DEFAULT	0x4
39	TXM_RO	0xb
40	TXM_RW	0x4
41	TXM_CPU_STACK	0x4
42	TXM_THREAD_STACK	0x4
43	TXM_ADDRESS_SPACE_TABLE	0x4
44	TXM_MALLOC_PAGE	0x4
45	TXM_FREE_LIST	0x4
46	TXM_SLAB_TRUST_CACHE	0x4
47	TXM_SLAB_PROFILE	0x4
48	TXM_SLAB_CODE_SIGNATURE	0x4
49	TXM_SLAB_CODE_REGION	0x4
50	TXM_SLAB_ADDRESS_SPACE	0x4
51	TXM_BUCKET_1024	0x4
52	TXM_BUCKET_2048	0x4
53	TXM_BUCKET_4096	0x4
54	TXM_BUCKET_8192	0x4
55	TXM_BULK_DATA	0x4
56	TXM_BULK_DATA_READ_ONLY	0xb
57	TXM_LOG	0x4
58	TXM_SEP_SECURE_CHANNEL	0x4
59	SK_DEFAULT	0xff
60	SK_SHARED_RO	0xb
61	SK_SHARED_RW	0x3
62	SK_IO	0xff

Table A.9: SPRR index retrieved for the specified SPTM frame type in SPTM `retype`.

A.14 Allowed SPTM Frame Types to Map Into a Table of the Specified Type

Table Type ID	Table Type Name	Entry Value	Allowed Frame Type to Insert
8	SPTM_KERNEL_ROOT_TABLE	0x280000	XNU_PAGE_TABLE, XNU_PAGE_TABLE_ROZONE
9	SPTM_PAGE_TABLE	0xFFFFFFFF	all types
17	XNU_USER_ROOT_TABLE	0x48000	XNU_PAGE_TABLE, XNU_PAGE_TABLE_COMMPAGE
18	XNU_SHARED_ROOT_TABLE	0x10000	XNU_PAGE_TABLE_SHARED
19	XNU_PAGE_TABLE	0x679E881	SPTM_RO, SPTM_XNU_CODE_DBG_RW, SPTM_PAGE_TABLE, SPTM_IOMMU_BOOTSTRAP, XNU_DEFAULT, XNU_RO, XNU_USER_DEBUG, XNU_USER_JIT, XNU_USER_ROOT_TABLE, XNU_SHARED_ROOT_TABLE, XNU_PAGE_TABLE_ROZONE, XNU_PAGE_TABLE_COMMPAGE
20	XNU_PAGE_TABLE_SHARED	0x2104801	SPTM_UNTYPED, XNU_DEFAULT, XNU_USER_EXEC, XNU_PAGE_TABLE_SHARED, XNU_IO
21	XNU_PAGE_TABLE_ROZONE	0x1200000	XNU_PAGE_TABLE_ROZONE, XNU_ROZONE
22	XNU_PAGE_TABLE_COMMPAGE	0x00004038	SPTM_RO, SPTM_CODE, SPTM_TXM_CODE, XNU_USER_EXEC
31	XNU_STAGE2_ROOT_TABLE	0x0	None
32	XNU_STAGE2_PAGE_TABLE	0x800	XNU_DEFAULT

Table A.10: Allowed SPTM frame types to map into a table of the specified type.

A.15 `Frameworks` and `PrivateFrameworks` Found in the `ExclaveKit` DMG

A.15.1 Frameworks

- AOPVoiceTriggerSecure.framework
- AVFAudio.framework
- Accelerate.framework
- AppleProxExclaveComponent.framework
- Common_ISP_EK_TBModule.framework
- CoreAudioTypes.framework
- CoreFoundation.framework
- DeviceTreeKit.framework
- EXDataLoader.framework
- EXMobileAssetLoader.framework
- EXSimpleFileIO.framework
- ExclaveFDRDecodeRawDataStoreKitComponent.framework

- ExclavesAudioDrivers.framework
- Foundation.framework
- IR_ISP_EK_TBModule.framework
- IsolatedAudioMeterClientExclaveComponent.framework
- IsolatedCoreAudioClientComponent.framework
- MobileAssetExclaveComponent.framework
- MobileGestalt.framework
- OSLogExclaves.framework
- RGB_ISP_EK_TBModule.framework
- SILManagerComponent.framework
- SharedMemory.framework
- SoundAnalysis.framework
- StackshotDelegateComponent.framework
- T8140_IR_ISP_EK_Component.framework
- T8140_RGB_ISP_EK_Component.framework
- TransitKit.framework
- Vision.framework
- VoiceTriggerCommon.framework
- VoiceTriggerSecure.framework

A.15.2 PrivateFrameworks

- ANEExclaveServices.framework
- ANSTCommon.framework
- AppleCVA.framework
- AppleSauce.framework
- AudioCaptureServer.framework
- AudioCaptureServerComponent.framework
- AudioDSP.framework
- AudioDSPControllerComponent.framework
- AudioDSPGraph.framework
- AudioDSPProcessorComponent.framework
- AudioToolboxCore.framework
- CollectionsInternal.framework
- CoreANSTKit.framework
- CoreAudioResources.framework
- CoreERClientKit.framework
- CoreMDClientKit.framework
- CoreMedia.framework
- CoreSpeechExclave.framework

- CoreSpeechExclaveComponent.framework
- CoreSpeechUtils.framework
- CoreVideo.framework
- DebugExfiltration.framework
- EXDisplayPipeSwapClient.framework
- EXSurface.framework
- Espresso.framework
- ExclaveCredentialManager.framework
- ExclaveFDRDecode.framework
- MIL.framework
- MLCompilerRuntime.framework
- MLCompilerServices.framework
- SISP_EK_AlgoModels.framework
- SecureVoiceTriggerAssets_exclavekit.framework
- ShareLibCommon_EK.framework
- ShazamKit.framework
- SpeakerRecognitionKit.framework
- SphinxProxTrustedFDR.framework
- T8140_CoreAAClientKit.framework
- T8140_ExclaveISPSharedLib_exclavekit.framework
- Tightbeam.framework
- VoiceTriggerEventProviderExclave.framework
- VoiceTriggerExclave.framework
- caulk.framework

A.16 Exclave Resource Type Specific Data Types

```
#define CONCLAVE_SERVICE_MAX 192

typedef struct {
    conclave_state_t       c_state;
    conclave_request_t     c_request;
    bool                   c_active_downcall;
    bool                   c_active_stopcall;
    bool                   c_active_detach;
    tb_client_connection_t c_control;
    task_t XNU_PTRAUTH_SIGNED_PTR("conclave.task") c_task;
    thread_t XNU_PTRAUTH_SIGNED_PTR("conclave.thread")
     ↪  c_downcall_thread;
    bitmap_t
     ↪  c_service_bitmap[BITMAP_LEN(CONCLAVE_SERVICE_MAX)];
} conclave_resource_t;

typedef struct {
    size_t sm_size;
    exclaves_buffer_perm_t sm_perm;
    char *sm_addr;
    sharedmemorybase_mapping_s sm_mapping;
    sharedmemorybase_segxnuaccess_s sm_client;
} shared_memory_resource_t;

typedef struct {
    /* how many times *this* sensor resource handle has been
     * used to call sensor_start */
    uint64_t s_startcount;
} sensor_resource_t;

typedef struct {
    struct klist notification_klist;
} exclaves_notification_t;
```

Listing A.5: Exclave resource type-specific data types as made available in `osfmk/kern/exclaves_resource.h` [32].

A.17 Exclaves Resources Grouped by Domain

Domain: `com.apple.audiomxd.conclave`

- `com.apple.audio.lpmic`
- `com.apple.audio.lpmic`
- `com.apple.audio.mic`
- `com.apple.audio.mic`
- `com.apple.audiomxd.AudioCaptureServer`
- `com.apple.audiomxd.MTDAudioDSPControl`
- `com.apple.audiomxd.PerceptionAudioDSPControl`
- `com.apple.audiomxd.SharedAudioDSPControl`

- com.apple.audiomxd.SiriAudioDSPControl
- com.apple.audiomxd.SoundAnalysisAudioDSPControl
- com.apple.inbound_buffer.hpmic16_injection
- com.apple.inbound_buffer.hpmic_injection
- com.apple.inbound_buffer.muted_talker_detection_ref_stream
- com.apple.inbound_buffer.muted_talker_detection_ref_stream
- com.apple.inbound_buffer.perception_ref_stream
- com.apple.inbound_buffer.shared_audiodsp_ref_stream
- com.apple.inbound_buffer.siri_ref_stream
- com.apple.inbound_buffer.siri_ref_stream
- com.apple.inbound_buffer.sound_analysis_ref_stream
- com.apple.inbound_buffer.sound_analysis_ref_stream
- com.apple.notification.audiocapture
- com.apple.notification.audiodsp
- com.apple.notification.audiodsp.analysis
- com.apple.sensors.mic

Domain: com.apple.backboardd.conclave

- com.apple.backboardd.AppleProxExclaveService
- com.apple.backboardd.ExclaveFDRDecodeRawDataStoreKitService
- com.apple.backboardd.SILManager

Domain: com.apple.cameracaptured.conclave

- com.apple.cameracaptured.ISPIRManager
- com.apple.cameracaptured.ISPIRService
- com.apple.cameracaptured.ISPRGBManager
- com.apple.cameracaptured.ISPRGBService

Domain: com.apple.conclave.mediaserverd

- com.apple.mediaserverd.ISPIRManager
- com.apple.mediaserverd.ISPIRService
- com.apple.mediaserverd.ISPRGBManager
- com.apple.mediaserverd.ISPRGBService

Domain: com.apple.conclave.test1

- com.apple.service.ExclavesCHelloServer

Domain: com.apple.corespeechd.conclave

- com.apple.audio.siri_history
- com.apple.audio.siri_history
- com.apple.corespeechd.SiriVoiceTriggerService
- com.apple.isolatedcoreaudioclient.service
- com.apple.sensors.mic

Domain: com.apple.darwin

- com.apple.service.ANEEngine
- com.apple.service.ANEExclave
- com.apple.service.ANEExclaveTestClient
- com.apple.service.ANEExclave_EDK
- com.apple.service.AudioDriver
- com.apple.service.AudioDriverLPMic
- com.apple.service.AudioDriverLPMic_EDK
- com.apple.service.AudioDriverVT
- com.apple.service.AudioDriverVT_EDK
- com.apple.service.AudioDriver_EDK
- com.apple.service.AudioSharedDARTMapper
- com.apple.service.AudioSharedDARTMapper_EDK
- com.apple.service.EXBrightCore
- com.apple.service.EXBrightCore_EDK
- com.apple.service.EXDisplayPipeDriver_EDK_xnuproxy
- com.apple.service.EXDisplayPipeDriver_xnuproxy
- com.apple.service.ExclaveDriverKit
- com.apple.service.ExclaveFDRDecodeTestClientExclave
- com.apple.service.ExclaveSEPManager
- com.apple.service.ExclaveSEPManager_EDK
- com.apple.service.ExclaveSEPTestClient
- com.apple.service.ExclaveTestService
- com.apple.service.HPMic16DMA
- com.apple.service.HPMic16DMA_EDK
- com.apple.service.HPMic16Device
- com.apple.service.HPMic16Device_EDK
- com.apple.service.HPMicAudioDriver
- com.apple.service.HPMicAudioDriver_EDK
- com.apple.service.HelloDrivers
- com.apple.service.HelloExclaveCxx
- com.apple.service.HelloExclaveKit
- com.apple.service.HelloStorage
- com.apple.service.SecureRTBuddyAOP
- com.apple.service.SecureRTBuddyAOP_EDK
- com.apple.service.SecureRTBuddyAOP_IOReporting
- com.apple.service.SecureRTBuddyDCP
- com.apple.service.SecureRTBuddyDCP_EDK
- com.apple.service.SecureRTBuddyDCP_IOReporting
- com.apple.service.TightbeamTestsServer
- com.apple.service.user_app
- com.apple.service.user_app2
- com.apple.service.user_app3

Domain: com.apple.exclavetestrunnerd.conclave

- com.apple.exclavetestrunnerd.service

Domain: com.apple.facekittestd.conclave

- com.apple.facekittestd.service

Domain: `com.apple.isolatedaudiometerclientd.conclave`

- `com.apple.isolatedaudiometerclientd.service`
- `com.apple.isolatedcoreaudioclient.service`

Domain: `com.apple.kernel`

- `com.apple.audio.lpmic`
- `com.apple.audio.lpmic`
- `com.apple.audio.mic`
- `com.apple.audio.mic`
- `com.apple.audio.siri_history`
- `com.apple.audio.siri_history`
- `com.apple.audio.test`
- `com.apple.audio.test`
- `com.apple.audiomxd.conclave`
- `com.apple.backboardd.conclave`
- `com.apple.cameracaptured.conclave`
- `com.apple.conclave.mediaserverd`
- `com.apple.conclave.test1`
- `com.apple.corespeechd.conclave`
- `com.apple.exclavetestrunnerd.conclave`
- `com.apple.facekittestd.conclave`
- `com.apple.inbound_buffer.exfiltration`
- `com.apple.inbound_buffer.hpmic16_injection`
- `com.apple.inbound_buffer.hpmic_injection`
- `com.apple.inbound_buffer.muted_talker_detection_ref_stream`
- `com.apple.inbound_buffer.siri_ref_stream`
- `com.apple.inbound_buffer.sound_analysis_ref_stream`
- `com.apple.isolatedaudiometerclientd.conclave`
- `com.apple.mobileassetd.conclave`
- `com.apple.named_buffer.1`
- `com.apple.named_buffer.1`
- `com.apple.named_buffer.10`
- `com.apple.named_buffer.112`
- `com.apple.named_buffer.113`
- `com.apple.named_buffer.16`
- `com.apple.named_buffer.2`
- `com.apple.named_buffer.2`
- `com.apple.named_buffer.3`
- `com.apple.named_buffer.32`
- `com.apple.named_buffer.36`
- `com.apple.named_buffer.4`
- `com.apple.named_buffer.41`
- `com.apple.named_buffer.6`
- `com.apple.named_buffer.7`
- `com.apple.notification.audiocapture`
- `com.apple.notification.audiodsp`
- `com.apple.notification.audiodsp.analysis`
- `com.apple.notification.hello`

- com.apple.outbound_buffer.exfiltration
- com.apple.perceptiond.conclave
- com.apple.securdled.conclave
- com.apple.sensors.cam
- com.apple.sensors.cam_alt_faceid
- com.apple.sensors.cam_alt_faceid_delayed
- com.apple.sensors.mic
- com.apple.service.ConclaveLauncherControl
- com.apple.service.ConclaveLauncherDebug
- com.apple.service.ExclaveIndicatorController
- com.apple.service.ExclaveKeyStoreTestClient
- com.apple.service.ExclavesCHelloServer
- com.apple.service.Exfiltration_xnuproxy
- com.apple.service.FrameMint
- com.apple.service.HelloDriverInterrupts
- com.apple.service.ISPCamEngine
- com.apple.service.ISPCamEngine_EDK
- com.apple.service.ISPEngine
- com.apple.service.ISPEngine_EDK
- com.apple.service.LogServer_xnuproxy
- com.apple.service.PanicInit
- com.apple.service.Stackshot
- com.apple.shareddspd.conclave
- com.apple.sharedmem.stackshotserver
- com.apple.soundanalysisd.conclave
- com.apple.storage.backend

Domain: com.apple.mobileassetd.conclave

- com.apple.mobileassetd.service

Domain: com.apple.perceptiond.conclave

- com.apple.isolatedcoreaudioclient.service
- com.apple.perceptiond.services

Domain: com.apple.securdled.conclave

- com.apple.securdled.service

Domain: com.apple.shareddspd.conclave

- com.apple.isolatedcoreaudioclient.service

Domain: com.apple.soundanalysisd.conclave

- com.apple.isolatedcoreaudioclient.service
- com.apple.soundanalysisd.service

A.18 Kernel Handling of Userspace Exclaves Endpoint Call

```
    switch (operation) {
    case EXCLAVES_CTL_OP_ENDPOINT_CALL: {
        if (name != MACH_PORT_NULL) {
            /* Only accept MACH_PORT_NULL for now */
            return KERN_INVALID_CAPABILITY;
        }
        if (ubuffer == USER_ADDR_NULL || usize == 0 ||
            usize != Exclaves_L4_IpcBuffer_Size) {
            return KERN_INVALID_ARGUMENT;
        }

        Exclaves_L4_IpcBuffer_t *ipcb;
        if ((error = exclaves_allocate_ipc_buffer((void**)&ipcb))) {
            return error;
        }
        assert(ipcb != NULL);
        if ((error = copyin(ubuffer, ipcb, usize))) {
            return error;
        }

        if (identifier >= CONCLAVE_SERVICE_MAX) {
            return KERN_INVALID_ARGUMENT;
        }

        /*
         * Verify that the service actually exists in the current
         * domain (only when the fallbacks are not enabled).
         */
        if (!exclaves_service_fallback &&
            !exclaves_conclave_has_service(task_get_conclave(task),
            identifier)) {
            return KERN_INVALID_ARGUMENT;
        }

        kr = exclaves_endpoint_call_internal(IPC_PORT_NULL,
         ↪  identifier);
        error = copyout(ipcb, ubuffer, usize);
        /*
         * Endpoint call to conclave may have trigger a stop upcall,
         * check if stop upcall completion handler needs to run.
         */
        task_stop_conclave_upcall_complete();
        if (error) {
            return error;
        }
        break;
    }
```

Listing A.6: Kernel handling of Exclave endpoint call invocation performed via `_exclaves_ctl_trap` with operation `EXCLAVES_CTL_OP_ENDPOINT_CALL`.

A.19 Xnuproxy available commands, from `cmd_to_string`

Also see `osfmk/kern/exclavs.c` l. 2311 [32].

- XNUPROXY_CMD_UNDEFINED

- XNUPROXY_CMD_SETUP
- XNUPROXY_CMD_CONTEXT_ALLOCATE
- XNUPROXY_CMD_CONTEXT_FREE
- XNUPROXY_CMD_NAMED_BUFFER_CREATE
- XNUPROXY_CMD_NAMED_BUFFER_DELETE
- XNUPROXY_CMD_RESOURCE_INFO
- XNUPROXY_CMD_AUDIO_BUFFER_CREATE
- XNUPROXY_CMD_AUDIO_BUFFER_COPYOUT
- XNUPROXY_CMD_AUDIO_BUFFER_DELETE
- XNUPROXY_CMD_SENSOR_START
- XNUPROXY_CMD_SENSOR_STOP
- XNUPROXY_CMD_SENSOR_STATUS
- XNUPROXY_CMD_DISPLAY_HEALTHCHECK_RATE
- XNUPROXY_CMD_NAMED_BUFFER_MAP
- XNUPROXY_CMD_NAMED_BUFFER_LAYOUT
- XNUPROXY_CMD_AUDIO_BUFFER_MAP
- XNUPROXY_CMD_AUDIO_BUFFER_LAYOUT
- XNUPROXY_CMD_REPORT_MEMORY_USAGE
- XNUPROXY_CMD_UPCALL_READY

A.20 Kernel `exclaves_xnu_proxy_send` Function

```
kern_return_t
exclaves_xnu_proxy_send(xnuproxy_msg_t *_msg, Exclaves_L4_Word_t
    *spawned)
{
    assert3p(_msg, !=, NULL);

    thread_t thread = current_thread();

    if (exclaves_xnu_proxy_msg_buffer == NULL) {
    return KERN_FAILURE;
    }

    kern_return_t kr = KERN_SUCCESS;
    xnuproxy_msg_t *msg = exclaves_xnu_proxy_msg_buffer;
    bool interrupted = false;

    lck_mtx_lock(&exclaves_xnu_proxy_lock);

    assert3u(thread->th_exclaves_state & TH_EXCLAVES_STATE_ANY, ==, 0);
    thread->th_exclaves_state |= TH_EXCLAVES_XNUPROXY;

    KDBG_RELEASE(MACHDBG_CODE(DBG_MACH_EXCLAVES,
        MACH_EXCLAVES_XNUPROXY)
        | DBG_FUNC_START, exclaves_xnu_proxy_scid, _msg->cmd);
```

```
23
24      *msg = *_msg;
25      msg->server_id = exclaves_xnu_proxy_scid;
26
27      os_atomic_store(&msg->status, XNUPROXY_MSG_STATUS_PROCESSING,
28      release);
29
30      while (os_atomic_load(&msg->status, relaxed) ==
31      XNUPROXY_MSG_STATUS_PROCESSING) {
32      exclaves_xnu_proxy_show_progress(in progress, msg);
33      kr = exclaves_scheduler_resume_scheduling_context(msg->server_id,
34          spawned, interrupted);
35      assert(kr == KERN_SUCCESS || kr == KERN_ABORTED);
36
37      /* A wait was interrupted. */
38      interrupted = kr == KERN_ABORTED;
39
40      if (NULL != exclaves_xnu_proxy_upcall_ipcb) {
41          if (XNUPROXY_MSG_STATUS_UPCALL ==
             ↪  XNUPROXY_CR_STATUS(exclaves_xnu_proxy_upcall_ipcb)) {
42                  xnuproxy_msg_status_t status = (xnuproxy_msg_status_t)
43                      XNUPROXY_CR_STATUS(exclaves_xnu_proxy_upcall_ipcb);
44                  (void) exclaves_handle_upcall(thread,
                     ↪  exclaves_xnu_proxy_upcall_ipcb,
45                      exclaves_xnu_proxy_scid, status);
46          }
47      }
48      }
49
50      if (os_atomic_load(&msg->status, acquire) ==
51      XNUPROXY_MSG_STATUS_NONE) {
52          exclaves_xnu_proxy_show_progress(complete, msg);
53      } else {
54          kr = KERN_FAILURE;
55          exclaves_xnu_proxy_show_error(msg);
56      }
57
58      *_msg = *msg;
59
60      KDBG_RELEASE(MACHDBG_CODE(DBG_MACH_EXCLAVES,
         ↪  MACH_EXCLAVES_XNUPROXY)
61      | DBG_FUNC_END);
62
63      thread->th_exclaves_state &= ~TH_EXCLAVES_XNUPROXY;
64      lck_mtx_unlock(&exclaves_xnu_proxy_lock);
65
66      return kr;
67  }
```

Listing A.7: `exclaves_xnu_proxy_send` function used for xnuproxy command invocation, from `osfmk/kern/exclaves.c` [32].

A.21 Tightbeam `__tb_connection_create_transport_for_endpoint` Function

```
void __tb_connection_create_transport_for_endpoint(tb_endpoint_t
 ↪  *tb_endpoint,int some_option)

{
  code *pcVar1;
  uint type;
  tb_endpoint_t *ptVar2;

  ptVar2 = tb_endpoint;
  _tb_endpoint_get_type(tb_endpoint);
  type = (uint)ptVar2;
  if (type == 1) {
    _tb_null_transport_create();
  }
  else if (type == 2) {
    if (some_option == 1) {
      _tb_mach_service_transport_create();
    }
    else {
      if (some_option != 0) goto LAB_25db24dac;
      _tb_mach_client_transport_create();
    }
  }
  else {
    if (type == 0) goto LAB_25db24dac;
    if ((some_option == 0) && ((type &
     ↪  0b11111111111111111111111111111110) == 4)) {
      _tb_eve_client_transport_create();
    }
    else if ((int)type < 10) {
      if (type == 8) {
        _tb_darwin_client_transport_create();
      }
      else {
        if (type != 9) goto LAB_25db24dac;
        if (some_option == 0) {
          _tb_unix_client_transport_create_with_endpoint();
        }
        else {
          if (some_option != 1) goto LAB_25db24dac;
          _tb_unix_service_transport_create_with_endpoint();
        }
      }
    }
    else if (type == 10) {
      if (some_option == 0) {
        _tb_delegated_client_transport_create();
      }
      else {
        if (some_option != 1) goto LAB_25db24dac;
        _tb_delegated_service_transport_create();
      }
    }
```

```
    else {
      if (type != 0xb) goto LAB_25db24dac;
      _tb_afk_transport_create();
    }
  }
  if (tb_endpoint != (tb_endpoint_t *)0x0) {
    return;
  }
LAB_25db24dac:
                    /* WARNING: Does not return */
  pcVar1 = (code *)SoftwareBreakpoint(1,0x25db24db0);
  (*pcVar1)();
}
```

Listing A.8: Tightbeam `__tb_connection_create_transport_for_endpoint` function.

A.22 `txm_dispatch_handler` Function in TXM

```
void txm_dispatch_handler
               (int selector,ulong param_2,undefined8
                ↪  param_3,undefined8 param_4,undefined8 param_5,
               undefined8 param_6)

{
  ulong uVar1;
  long lVar2;
  undefined8 uVar3;
  long lVar4;
  uint uVar5;
  ulong local_70 [4];

  lVar2 = FUN_fffffff017024e44();
  local_70[3] = 1;
  if (DAT_fffffff017060380 != '\0') {
    uVar3 = 0xa0;
LAB_fffffff017022d1c:
                    /* WARNING: Subroutine does not return */
    FUN_fffffff017021bd8(uVar3,0);
  }
  uVar1 = (ulong)(local_70 + 3) & 0xfffffffffffffc000;
  if (uVar1 == 0) {
    uVar3 = 0x40;
    goto LAB_fffffff017022d1c;
  }
  if ((ulong *)0xfffffffffffffbfff < local_70 + 3) {
    uVar3 = 0x42;
    goto LAB_fffffff017022d1c;
  }
  local_70[2] = 0x4000;
  local_70[1] = 0x4000;
  local_70[0] = uVar1;
  FUN_fffffff017024790(local_70,0x2a);
  *(undefined8 *)(lVar2 + 0x18) = 0;
  if (0x32 < selector - 1U) {
```

```
    uVar3 = 0xa1;
    goto LAB_fffffff017022d1c;
  }
  uVar5 = (uint)param_2;
  switch(selector) {
  default:
    lVar4 = FUN_fffffff017024e44();
    *(undefined8 *)(lVar4 + 0x18) = 3;
    *(undefined8 *)(lVar4 + 0x20) = DAT_fffffff017010500;
    *(undefined4 **)(lVar4 + 0x28) = &DAT_fffffff017060384;
    *(undefined4 **)(lVar4 + 0x30) = &DAT_fffffff017060388;
  case 0x1f:
code_r0xfffffff017023048:
    local_70[3] = 0;
    break;
  case 2:
    FUN_fffffff017023258();
    goto code_r0xfffffff017023048;
  case 3:
    lVar4 = FUN_fffffff017024e44();
    *(undefined8 *)(lVar4 + 0x18) = 4;
    *(undefined8 **)(lVar4 + 0x20) = &DAT_fffffff017010590;
    *(ulong *)(lVar4 + 0x28) = (ulong)DAT_fffffff0170105b8;
    *(undefined8 *)(lVar4 + 0x30) = 0;
    *(undefined8 *)(lVar4 + 0x38) = 0;
    goto code_r0xfffffff017023048;
  case 4:
    lVar4 = FUN_fffffff017024e44();
    *(undefined8 *)(lVar4 + 0x18) = 1;
    uVar3 = get_build_string();
    *(undefined8 *)(lVar4 + 0x20) = uVar3;
    goto code_r0xfffffff017023048;
  case 5:
    FUN_fffffff01701be44();
    goto code_r0xfffffff017023048;
  case 6:
    local_70[3] = FUN_fffffff0170233d0();
    break;
  case 7:
    local_70[3] = FUN_fffffff0170232c0();
    break;
  case 8:
    local_70[3] = FUN_fffffff01701ba98();
    break;
  case 9:
    FUN_fffffff017023348(param_2);
    goto code_r0xfffffff017023048;
  case 10:
    FUN_fffffff017023464(param_2);
    goto code_r0xfffffff017023048;
  case 0xb:
    local_70[3] = FUN_fffffff0170234c0();
    break;
  case 0xc:
    local_70[3] = FUN_fffffff01702351c(uVar5 &
     ↪  0xff,param_3,param_4,param_5,param_6);
    break;
```

```
92    case 0xd:
93      if (param_2 + 0x10 < param_2) goto LAB_fffffff01702317c;
94      local_70[3] = FUN_fffffff01701e3f4(param_2);
95      break;
96    case 0xe:
97      local_70[3] = FUN_fffffff017023638(uVar5 & 0xff,param_3);
98      break;
99    case 0xf:
100     local_70[3] = FUN_fffffff017023720(param_2);
101     break;
102   case 0x10:
103     local_70[3] = FUN_fffffff0170237f4(param_2);
104     break;
105   case 0x11:
106     local_70[3] = FUN_fffffff017023898(param_2,param_3);
107     break;
108   case 0x12:
109     local_70[3] = FUN_fffffff017023958(param_2,param_3,param_4);
110     break;
111   case 0x13:
112     local_70[3] = FUN_fffffff017023a2c(param_2);
113     break;
114   case 0x14:
115     local_70[3] = FUN_fffffff017023a88(param_2,param_3);
116     break;
117   case 0x15:
118     local_70[3] = FUN_fffffff017023ad0(param_2);
119     break;
120   case 0x16:
121     local_70[3] = FUN_fffffff017023b04(param_2,param_3,param_4);
122     break;
123   case 0x17:
124     local_70[3] = FUN_fffffff017023bdc(param_2);
125     break;
126   case 0x18:
127     FUN_fffffff01701f684(param_2);
128     local_70[3] = FUN_fffffff01701c728();
129     break;
130   case 0x19:
131     local_70[3] = FUN_fffffff017023c38(param_2);
132     break;
133   case 0x1a:
134     if (param_2 + 0x61 < param_2) goto LAB_fffffff01702317c;
135     local_70[3] = FUN_fffffff01701d398(param_2);
136     break;
137   case 0x1b:
138     local_70[3] = FUN_fffffff017023c98();
139     break;
140   case 0x1c:
141     if (param_2 + 0x14 < param_2) goto LAB_fffffff01702317c;
142     local_70[3] = FUN_fffffff01701d430(param_2);
143     break;
144   case 0x1d:
145     if (param_2 + 0x14 < param_2) goto LAB_fffffff01702317c;
146     local_70[3] = FUN_fffffff01701d5b4(param_2);
147     break;
148   case 0x1e:
```

```
149       local_70[3] = FUN_fffffff017023ce0(param_2);
150       break;
151     case 0x20:
152       local_70[3] = FUN_fffffff017023d4c(param_2);
153       break;
154     case 0x21:
155       uVar3 = FUN_fffffff01701f684(param_2);
156       local_70[3] = FUN_fffffff01701cd70(uVar3,param_3);
157       break;
158     case 0x22:
159       local_70[3] = FUN_fffffff017023da4(param_2);
160       break;
161     case 0x23:
162       local_70[3] = FUN_fffffff01701a268(param_2,param_3);
163       break;
164     case 0x24:
165       local_70[3] = FUN_fffffff017023e34(uVar5 & 0xffff,param_3);
166       break;
167     case 0x25:
168       FUN_fffffff017020280(param_2);
169       local_70[3] = FUN_fffffff01701a3b0();
170       break;
171     case 0x26:
172       local_70[3] = associate_code_signature(param_2,param_3,param_4,para
          ↪  m_5,param_6);
173       break;
174     case 0x27:
175       local_70[3] = FUN_fffffff017023f08(param_2);
176       break;
177     case 0x28:
178       uVar3 = FUN_fffffff017020280(param_2);
179       local_70[3] = FUN_fffffff01701b1fc(uVar3,param_3,param_4);
180       break;
181     case 0x29:
182       FUN_fffffff017020280(param_2);
183       local_70[3] = allow_invalid_code?();
184       break;
185     case 0x2a:
186       uVar3 = FUN_fffffff017020280(param_2);
187       local_70[3] = FUN_fffffff01701b328(uVar3,param_3,param_4);
188       break;
189     case 0x2b:
190       local_70[3] = FUN_fffffff017023f54(param_2);
191       break;
192     case 0x2c:
193       local_70[3] = FUN_fffffff017023fe0(param_2);
194       break;
195     case 0x2d:
196       local_70[3] = FUN_fffffff017024038(param_2,param_3,param_4);
197       break;
198     case 0x2e:
199     case 0x2f:
200     case 0x30:
201     case 0x31:
202     case 0x32:
203     case 0x33:
204       local_70[3] = 0x26;
```

```
205     }
206     *(ulong *)(lVar2 + 8) = local_70[3];
207     FUN_fffffff017022a30();
208   LAB_fffffff01702317c:
209                       /* WARNING: Subroutine does not return */
210     FUN_fffffff017021d2c(0x19);
```

Listing A.9: `txm_dispatch_handler` found in the TXM binary, invoked on TXM calls from XNU to perform actual function handling based on a provided selector.

A.23 Allowed SPTM Frame Type Transitions

Type	Allowed retypes
SPTM_UNTYPED	SPTM_UNTYPED, SPTM_UNUSED, SPTM_DEFAULT, SPTM_RO, SPTM_CODE, SPTM_TXM_CODE, SPTM_XNU_CODE, SPTM_XNU_CODE_DBG_RW, SPTM_KERNEL_ROOT_TABLE, SPTM_PAGE_TABLE, SPTM_IOMMU_BOOTSTRAP, XNU_DEFAULT, XNU_RO, XNU_RO_DBG_RW, XNU_USER_EXEC, XNU_USER_DEBUG, XNU_USER_JIT, XNU_USER_ROOT_TABLE, XNU_SHARED_ROOT_TABLE, XNU_PAGE_TABLE, XNU_PAGE_TABLE_SHARED, XNU_PAGE_TABLE_ROZONE, XNU_PAGE_TABLE_COMMPAGE, XNU_IOMMU, XNU_ROZONE, XNU_IO, XNU_PROTECTED_IO, XNU_COMMPAGE_RW, XNU_COMMPAGE_RO, XNU_COMMPAGE_RX, XNU_TAG_STORAGE, XNU_STAGE2_ROOT_TABLE, XNU_STAGE2_PAGE_TABLE, XNU_KERNEL_RESTRICTED, XNU_RESERVED_1, XNU_RESERVED_2, XNU_RESTRICTED_IO, XNU_RESTRICTED_IO_TELEMETRY, TXM_DEFAULT, TXM_RO, TXM_RW, TXM_CPU_STACK, TXM_THREAD_STACK, TXM_ADDRESS_SPACE_TABLE, TXM_MALLOC_PAGE, TXM_FREE_LIST, TXM_SLAB_TRUST_CACHE, TXM_SLAB_PROFILE, TXM_SLAB_CODE_SIGNATURE, TXM_SLAB_CODE_REGION, TXM_SLAB_ADDRESS_SPACE, TXM_BUCKET_1024, TXM_BUCKET_2048, TXM_BUCKET_4096, TXM_BUCKET_8192, TXM_BULK_DATA, TXM_BULK_DATA_READ_ONLY, TXM_LOG, TXM_SEP_SECURE_CHANNEL, SK_DEFAULT, SK_SHARED_RO, SK_SHARED_RW, SK_IO
SPTM_UNUSED	
SPTM_DEFAULT	
SPTM_RO	
SPTM_CODE	
SPTM_TXM_CODE	
SPTM_XNU_CODE	
SPTM_XNU_CODE_DBG_RW	
SPTM_KERNEL_ROOT_TABLE	
SPTM_PAGE_TABLE	
SPTM_IOMMU_BOOTSTRAP	

Type	Allowed retypes
XNU_DEFAULT	SPTM_UNUSED, XNU_DEFAULT, XNU_USER_EXEC, XNU_USER_DEBUG, XNU_USER_JIT, XNU_USER_ROOT_TABLE, XNU_PAGE_TABLE, XNU_PAGE_TABLE_SHARED, XNU_PAGE_TABLE_ROZONE, XNU_PAGE_TABLE_COMMPAGE, XNU_IOMMU, XNU_ROZONE, XNU_COMMPAGE_RW, XNU_COMMPAGE_RO, XNU_COMMPAGE_RX, XNU_STAGE2_ROOT_TABLE, XNU_STAGE2_PAGE_TABLE, XNU_KERNEL_RESTRICTED, XNU_RESERVED_2, TXM_DEFAULT SK_DEFAULT
XNU_RO	
XNU_RO_DBG_RW	
XNU_USER_EXEC	XNU_DEFAULT
XNU_USER_DEBUG	XNU_DEFAULT
XNU_USER_JIT	XNU_DEFAULT
XNU_USER_ROOT_TABLE	XNU_USER_DEFAULT, XNU_SHARED_ROOT_TABLE
XNU_SHARED_ROOT_TABLE	XNU_DEFAULT
XNU_PAGE_TABLE	XNU_DEFAULT
XNU_PAGE_TABLE_SHARED	XNU_DEFAULT
XNU_PAGE_TABLE_ROZONE	
XNU_PAGE_TABLE_COMMPAGE	
XNU_IOMMU	XNU_DEFAULT
XNU_ROZONE	XNU_DEFAULT
XNU_IO	
XNU_PROTECTED_IO	
XNU_COMMPAGE_RW	
XNU_COMMPAGE_RO	
XNU_COMMPAGE_RX	
XNU_TAG_STORAGE	
XNU_STAGE2_ROOT_TABLE	XNU_DEFAULT
XNU_STAGE2_PAGE_TABLE	XNU_DEFAULT
XNU_KERNEL_RESTRICTED	XNU_DEFAULT, XNU_KERNEL_RESTRICTED
XNU_RESERVED_1	
XNU_RESERVED_2	XNU_DEFAULT
XNU_RESTRICTED_IO	
XNU_RESTRICTED_IO_TELEMETRY	
TXM_DEFAULT	TXM_FREE_LIST, TXM_BULK_DATA, TXM_BULK_DATA_READ_ONLY
TXM_RO	
TXM_RW	
TXM_CPU_STACK	
TXM_THREAD_STACK	
TXM_ADDRESS_SPACE_TABLE	
TXM_MALLOC_PAGE	

Type	Allowed retypes
TXM_FREE_LIST	XNU_DEFAULT, TXM_SLAB_TRUST_CACHE, TXM_SLAB_PROFILE, TXM_SLAB_CODE_SIGNATURE, TXM_SLAB_CODE_REGION, TXM_SLAB_ADDRESS_SPACE, TXM_BUCKET_1024, TXM_BUCKET_2048, TXM_BUCKET_4096, TXM_BUCKET_8192
TXM_SLAB_TRUST_CACHE	
TXM_SLAB_PROFILE	
TXM_SLAB_CODE_SIGNATURE	
TXM_SLAB_CODE_REGION	
TXM_SLAB_ADDRESS_SPACE	
TXM_BUCKET_1024	
TXM_BUCKET_2048	
TXM_BUCKET_4096	
TXM_BUCKET_8192	
TXM_BULK_DATA	XNU_DEFAULT, TXM_BULK_DATA_READ_ONLY
TXM_BULK_DATA_READ_ONLY	XNU_DEFAULT
TXM_LOG	
TXM_SEP_SECURE_CHANNEL	
SK_DEFAULT	XNU_DEFAULT, SK_SHARED_RO, SK_SHARED_RW
SK_SHARED_RO	SK_DEFAULT, SK_SHARED_RW
SK_SHARED_RW	SK_DEFAULT, SK_SHARED_RO
SK_IO	

Table A.11: The table lists allowed frame retypings extracted from the SPTM binary based on conditional checks in its `retype` operation. The type column shows the current frame type, while allowed retypes shows valid new frame types retype to.

A.24 Type-Specific Retype Functions

Type	Type Name	type_out_function	type_in_function
8	SPTM_KERNEL_ROOT_TABLE	FUN_ffffffo270bacdo	FUN_ffffffo270baffc
11	XNU_DEFAULT	FUN_ffffffo270ba9e8	FUN_ffffffo270bacb8
14	XNU_USER_EXEC	FUN_ffffffo270ba918	-
15	XNU_USER_DEBUG	FUN_ffffffo270ba918	-
16	XNU_USER_JIT	FUN_ffffffo270ba918	-
17	XNU_USER_ROOT_TABLE	FUN_ffffffo270bacdo	FUN_ffffffo270baffc
18	XNU_SHARED_ROOT_TABLE	FUN_ffffffo270bacdo	FUN_ffffffo270baffc
19	XNU_PAGE_TABLE	FUN_ffffffo270ba7c4	FUN_ffffffo270ba8a8
20	XNU_PAGE_TABLE_SHARED	FUN_ffffffo270ba7c4	FUN_ffffffo270ba8a8
21	XNU_PAGE_TABLE_ROZONE	-	FUN_ffffffo270ba8a8
22	XNU_PAGE_TABLE_COMMPAGE	-	FUN_ffffffo270ba8a8
23	XNU_IOMMU	FUN_ffffffo270ba540	FUN_ffffffo270ba66c
24	XNU_ROZONE	FUN_ffffffo270ba45c	-
31	XNU_STAGE2_ROOT_TABLE	FUN_ffffffo270bacdo	FUN_ffffffo270baffc
32	XNU_STAGE2_PAGE_TABLE	FUN_ffffffo270ba8a8	FUN_ffffffo270ba7c4

Continued on next page

Type	Type Name	type_out_function	type_in_function
33	XNU_KERNEL_RESTRICTED	FUN_ffffffo270ba9e8	-
35	XNU_RESERVED_2	FUN_ffffffo270ba9e8	FUN_ffffffo270bacb8
59	SK_DEFAULT	FUN_ffffffo270ba318	-
60	SK_SHARED_RO	FUN_ffffffo270ba318	-
61	SK_SHARED_RW	FUN_ffffffo270ba318	

Table A.12: Type-specific `retype_in` and `retype_out` functions called during an SPTM `retype` operation.

A.25 SPTM Exception Handler Function `synchronous_exception_handler_from_lower`

```
1  *************************************************************
2  *                          FUNCTION                         *
3  *************************************************************
4              undefined synchronous_exception_handler_from_lower()
5  fffffff027081e38 9f 40 00 d5     msr        PState.PAN,#0x0
6  fffffff027081e3c e8 27 3f a9     stp        x8,x9,[sp, #local_10]
7  fffffff027081e40 28 fb 3e d5     mrs        x8,sreg(0x3, 0x6, c0xf,
    ↪  c0xb, 0x1)
8  fffffff027081e44 08 01 08 aa     orr        x8,x8,x8
9  fffffff027081e48 08 41 11 91     add        x8,x8,#0x450
10 fffffff027081e4c 08 11 40 f9     ldr        x8,[x8, #0x20]
11 fffffff027081e50 ff 63 28 eb     cmp        sp,x8
12                    LAB_fffffff027081e54
13 fffffff027081e54 01 00 00 54     b.ne       LAB_fffffff027081e54
14 fffffff027081e58 08 11 3c d5     mrs        x8,hcr_el2
15 fffffff027081e5c 08 6d 40 92     and        x8,x8,#0xffffffff
16 fffffff027081e60 08 95 65 92     and        x8,x8,#-0x7fffffff
17 fffffff027081e64 1f 05 00 f1     cmp        x8,#0x1
18 fffffff027081e68 00 01 00 54     b.eq       LAB_fffffff027081e88
19                    GL1 Exception Syndrome Register
20 fffffff027081e6c a8 fa 3e d5     mrs        x8,sreg(0x3, 0x6, c0xf,
    ↪  c0xa, 0x5)
21 fffffff027081e70 08 7d 5a d3     ubfx       x8,x8,#0x1a,#0x6
22 fffffff027081e74 1f 55 00 f1     cmp        x8,#0x15
23 fffffff027081e78 80 08 00 54     b.eq       SVC_HANDLER
24 fffffff027081e7c 1f 59 00 f1     cmp        x8,#0x16
25 fffffff027081e80 e0 0f 00 54     b.eq       LAB_fffffff02708207c
26 fffffff027081e84 8b 00 00 14     b          LAB_fffffff0270820b0
```

Listing A.10: SPTM synchronous exception handler function `synchronous_exception_handler_from_lower`.

A.26 SPTM SVC #0 Handler

```
                        SVC_0_HANDLER
fffffff027081ff0 09 9e 60 d3     ubfx        x9,x16,#0x20,#0x8
fffffff027081ff4 aa 1f 80 d2     mov         x10,#0xfd
fffffff027081ff8 3f 01 0a eb     cmp         x9,x10
fffffff027081ffc 41 00 00 54     b.ne        SVC_0_not_0xfd
fffffff027082000 18 01 00 14     b           SVC_0_is_0xfd
                 undefined SVC_0_is_0xfd()

                        SVC_0_not_0xfd
fffffff027082004 ca 1f 80 d2     mov         x10,#0xfe
fffffff027082008 3f 01 0a eb     cmp         x9,x10
fffffff02708200c 01 03 00 54     b.ne        SVC_0_not_0xfe
fffffff027082010 28 fb 3e d5     mrs         x8,sreg(0x3, 0x6, c0xf,
                 c0xb, 0x1)
fffffff027082014 08 01 08 aa     orr         x8,x8,x8
fffffff027082018 08 41 11 91     add         x8,x8,#0x450
fffffff02708201c 0a 41 15 91     add         x10,x8,#0x550
fffffff027082020 49 01 40 f9     ldr         x9,[x10]
fffffff027082024 a9 f1 1e d5     msr         sreg(0x3, 0x6, c0xf, c0x1,
                 0x5),x9
fffffff027082028 49 05 40 f9     ldr         x9,[x10, #0x8]
fffffff02708202c 49 02 10 d5     msr         mdscr_el1,x9
fffffff027082030 49 09 40 f9     ldr         x9,[x10, #0x10]
fffffff027082034 a9 f1 19 d5     msr         sreg(0x3, 0x1, c0xf, c0x1,
                 0x5),x9
fffffff027082038 49 0d 40 f9     ldr         x9,[x10, #0x18]
fffffff02708203c 89 f0 1c d5     msr         sreg(0x3, 0x4, c0xf, c0x0,
                 0x4),x9
fffffff027082040 49 11 40 f9     ldr         x9,[x10, #0x20]
fffffff027082044 09 10 18 d5     msr         sctlr_el1,x9
fffffff027082048 49 15 40 f9     ldr         x9,[x10, #0x28]
fffffff02708204c 49 20 18 d5     msr         tcr_el1,x9
fffffff027082050 bf 40 00 d5     msr         PState.SP,#0x0
fffffff027082054 01 00 80 d2     mov         x1,#0x0
fffffff027082058 22 00 80 d2     mov         x2,#0x1
fffffff02708205c c1 fc ff 17     b           FUN_fffffff027081360

...
                        SVC_0_is_0xfd
fffffff027082460 28 fb 3e d5     mrs         x8,sreg(0x3, 0x6, c0xf,
                 c0xb, 0x1)
fffffff027082464 08 01 08 aa     orr         x8,x8,x8
fffffff027082468 08 41 11 91     add         x8,x8,#0x450
fffffff02708246c 09 11 40 f9     ldr         x9,[x8, #0x20]
fffffff027082470 3f 01 00 91     mov         sp,x9
fffffff027082474 1e 00 00 90     adrp        x30,-0xfd8f7e000
fffffff027082478 de 93 16 91     add         x30,x30,#0x5a4
fffffff02708247c 1d 00 80 d2     mov         x29,#0x0

...

                        SVC_0_not_0xfe
fffffff02708206c ea 1f 80 d2     mov         x10,#0xff
fffffff027082070 3f 01 0a eb     cmp         x9,x10
```

```
fffffff027082074 a1 19 00 54     b.ne        SVC_0_not_0xff
fffffff027082078 1d 01 00 14     b           SVC_0_is_0xff

...

                    SVC_0_not_0xff
fffffff0270823a8 28 fb 3e d5     mrs         x8,sreg(0x3, 0x6, c0xf,
 ↪  c0xb, 0x1)
fffffff0270823ac 08 01 08 aa     orr         x8,x8,x8
fffffff0270823b0 08 41 11 91     add         x8,x8,#0x450
fffffff0270823b4 09 11 40 f9     ldr         x9,[x8, #0x20]
fffffff0270823b8 3f 01 00 91     mov         sp,x9
fffffff0270823bc 09 01 01 91     add         x9,x8,#0x40
fffffff0270823c0 20 05 00 a9     stp         x0,x1,[x9]
fffffff0270823c4 22 0d 01 a9     stp         x2,x3,[x9, #0x10]
fffffff0270823c8 24 15 02 a9     stp         x4,x5,[x9, #0x20]
fffffff0270823cc 26 1d 03 a9     stp         x6,x7,[x9, #0x30]
fffffff0270823d0 0a 1d 40 f9     ldr         x10,[x8, #0x38]
fffffff0270823d4 5f 05 00 f1     cmp         x10,#0x1

...

                    SVC_0_is_0xff
fffffff0270824ec 28 fb 3e d5     mrs         x8,sreg(0x3, 0x6, c0xf,
 ↪  c0xb, 0x1)
fffffff0270824f0 08 01 08 aa     orr         x8,x8,x8
fffffff0270824f4 08 41 11 91     add         x8,x8,#0x450
fffffff0270824f8 09 11 40 f9     ldr         x9,[x8, #0x20]
fffffff0270824fc 3f 01 00 91     mov         sp,x9
fffffff027082500 1e 00 00 90     adrp        x30,-0xfd8f7e000
fffffff027082504 de 93 16 91     add         x30,x30,#0x5a4
fffffff027082508 1d 00 80 d2     mov         x29,#0x0
```

Listing A.11: SPTM SVC #0 handler.

A.27 Function Signatures for User-Space Available Exclaves Functions

Taken from `libsyscall/wrappers/exclaves.c`.

```
kern_return_t exclaves_endpoint_call(mach_port_t port, exclaves_id_t
 ↪  endpoint_id,
    mach_vm_address_t msg_buffer, mach_vm_size_t size, exclaves_tag_t
     ↪  *tag,
    exclaves_error_t *error)

kern_return_t exclaves_outbound_buffer_create(mach_port_t port, const
 ↪  char *buffer_name,
    mach_vm_size_t size, mach_port_t *out_outbound_buffer_port)

kern_return_t exclaves_outbound_buffer_copyout(mach_port_t
 ↪  outbound_buffer_port,
    mach_vm_address_t dst_buffer, mach_vm_size_t size1, mach_vm_size_t
     ↪  offset1,
```

```
    mach_vm_size_t size2, mach_vm_size_t offset2)

kern_return_t exclaves_inbound_buffer_create(mach_port_t port, const char *buffer_name,
    mach_vm_size_t size, mach_port_t *out_inbound_buffer_port)

kern_return_t exclaves_inbound_buffer_copyin(mach_port_t inbound_buffer_port,
    mach_vm_address_t src_buffer, mach_vm_size_t size1, mach_vm_size_t offset1,
    mach_vm_size_t size2, mach_vm_size_t offset2)

kern_return_t exclaves_named_buffer_create(mach_port_t port, exclaves_id_t buffer_id,
    mach_vm_size_t size, mach_port_t *out_named_buffer_port)

kern_return_t exclaves_named_buffer_copyin(mach_port_t named_buffer_port,
    mach_vm_address_t src_buffer, mach_vm_size_t size, mach_vm_size_t offset)

kern_return_t exclaves_named_buffer_copyout(mach_port_t named_buffer_port,
    mach_vm_address_t dst_buffer, mach_vm_size_t size, mach_vm_size_t offset)

kern_return_t exclaves_launch_conclave(mach_port_t port, void *arg1,
    uint64_t arg2)

kern_return_t exclaves_lookup_service(mach_port_t port, const char *name,
    exclaves_id_t *resource_id)

kern_return_t exclaves_boot(mach_port_t port, exclaves_boot_stage_t stage)

kern_return_t exclaves_audio_buffer_create(mach_port_t port, const char *buffer_name,
    mach_vm_size_t size, mach_port_t* out_audio_buffer_port)

kern_return_t exclaves_audio_buffer_copyout(mach_port_t audio_buffer_port,
    mach_vm_address_t dst_buffer,
    mach_vm_size_t size1, mach_vm_size_t offset1,
    mach_vm_size_t size2, mach_vm_size_t offset2)

kern_return_t exclaves_sensor_create(mach_port_t port, const char *sensor_name,
    mach_port_t *sensor_port)

kern_return_t exclaves_sensor_start(mach_port_t sensor_port, uint64_t flags,
    exclaves_sensor_status_t *sensor_status)

kern_return_t exclaves_sensor_stop(mach_port_t sensor_port, uint64_t flags,
    exclaves_sensor_status_t *sensor_status)
```

```
52
53 kern_return_t exclaves_sensor_status(mach_port_t sensor_port, uint64_t
    ↪ flags,
54     exclaves_sensor_status_t *sensor_status)
55
56 kern_return_t exclaves_notification_create(__unused mach_port_t port,
    ↪ const char *name,
57     uint64_t *notification_id)
```

Listing A.12: User-space Exclave function signatures.

A.28 Ghidra Jython Script Extracting Exclave Resources and Corresponding Domains

This script extracts Exclave resources and domains from the `sharedcache` binary, grouping them by domains.

```
1  # -*- coding: utf-8 -*-
2  #@author Moritz Steffin
3  #@category Analysis
4
5  """
6  Ghidra script to extract and group resource entries by domain.
7
8  Each entry:
9  +0x00 : resource_name (C string)
10 +0x80 : domain_name (C string)
11 Size  : 0x110 bytes
12 Count : 0xa3 entries
13
14 Output: Grouped text file
15 """
16
17 ENTRY_COUNT = 0xa3
18 ENTRY_SIZE = 0x110
19 RESOURCE_NAME_OFFSET = 0x0
20 DOMAIN_NAME_OFFSET = 0x80
21 STRING_READ_MAXLEN = 256
22
23 def askBaseAddress():
24     input_str = askString("Base Address", "Enter the resource table
    ↪ base address (hex):")
25     if input_str is None:
26         exit(0)
27     input_str = input_str.strip()
28     if input_str.startswith("0x") or input_str.startswith("0X"):
29         input_str = input_str[2:]
30     try:
31         return toAddr(int(input_str, 16))
32     except:
33         popup("Invalid address: {}".format(input_str))
34         exit(1)
35
36 def read_c_string(addr, maxlen=256):
```

```
    result = []
    for i in range(maxlen):
        b = getByte(addr.add(i)) & 0xFF
        if b == 0:
            break
        if 32 <= b <= 126:
            result.append(chr(b))
        else:
            result.append('.')
    return ''.join(result)

def main():
    base_addr = askBaseAddress()
    print("[*] Reading resource table at {}".format(base_addr))

    # Build domain -> [resources] map
    domain_map = {}

    for i in range(ENTRY_COUNT):
        entry_addr = base_addr.add(i * ENTRY_SIZE)
        resource_addr = entry_addr.add(RESOURCE_NAME_OFFSET)
        domain_addr = entry_addr.add(DOMAIN_NAME_OFFSET)

        resource_name = read_c_string(resource_addr,
         ↪  STRING_READ_MAXLEN)
        domain_name = read_c_string(domain_addr, STRING_READ_MAXLEN)

        if not resource_name:
            resource_name = "<undefined>"
        if not domain_name:
            domain_name = "<undefined>"

        if domain_name not in domain_map:
            domain_map[domain_name] = []
        domain_map[domain_name].append(resource_name)

    # Ask where to save
    output_file = askFile("Save Text File", "Save").absolutePath

    with open(output_file, "w") as f:
        for domain in sorted(domain_map.keys()):
            f.write("Domain: {}\n".format(domain))
            for res in sorted(domain_map[domain]):
                f.write("  - {}\n".format(res))
            f.write("\n")

    popup("Done!\n\nExtracted and grouped {} domains.\nSaved
     ↪  to:\n{}".format(len(domain_map), output_file))

if __name__ == "__main__":
    main()
```

Listing A.13: Ghidra Script for Extracting Exclave Resources and Grouping them by Corresponding Domains from the sharedcache Binary.

A.29 Kernel `tb_connection_send_query` Function

```
ulong _tb_connection_send_query
                (tb_connection_t *connection,tb_message
                 ↪  *message,undefined8 *response,
                undefined8 some_sort_of_state)

{
  int iVar1;
  int splitRequired?;
  long tb_message_tpt_buffer;
  ulong send_return;
  long oberserver;
  undefined8 *puVar2;
  tb_message *message_00;
  tb_bufffer *tpt_buffer;
  undefined *puVar3;
  tb_transport_t *transport;
  undefined8 *puVar4;
  ulong uVar5;
  long local_70;

  if (message->msg_state == TB_MESSAGE_STATE_READY) {
    if (message->disposition != TB_MESSAGE_DISPOSITION_QUERY) goto
     ↪  LAB_ffffff0088ede10;
    iVar1 = tb_message_check_connection_identifier(message,connection);
    if (iVar1 == 0) {
      return TB_ERROR;
    }
    tb_message_set_state(message,TB_MESSAGE_STATE_SENT);
    transport = (tb_transport_t *)connection->transport;
    tb_message_tpt_buffer = tb_message_get_transport_buffer(message);
    if (((uint)some_sort_of_state >> 1 & 1) == 0) {
      *(ushort *)(tb_message_tpt_buffer + 0x2a) = *(ushort
       ↪  *)(tb_message_tpt_buffer + 0x2a) | 0x10;
    }
    splitRequired? =
         _tb_message_splitter_split_required
                   (transport,*(undefined8 *)(tb_message_tpt_buffer +
                    ↪  0x18));
    if (splitRequired? == 0) {
      if (((long *)connection->observers != (long *)0x0) &&
         (oberserver = *(long *)connection->observers, oberserver !=
          ↪  0)) {
        (**(code **)(oberserver + 0x10))
        (oberserver + 0x10,oberserver,transport,message,response,some_s
         ↪  ort_of_state);
      }
      send_return = tb_transport_send_message(transport);
    }
    else {
      send_return = _tb_message_splitter_send
            (connection,transport,message,response,some_sort_of_state);
    }
    if ((int)send_return != 0) {
      return send_return;
```

```
    }
    uVar5 = (ulong)((*(ushort *)(tb_message_tpt_buffer + 0x2a) & 8) >>
     ↪  1);
    if (((uint)some_sort_of_state >> 1 & 1) == 0) {
      return uVar5;
    }
    if ((*(ushort *)(tb_message_tpt_buffer + 0x2a) & 8) != 0) {
      return uVar5;
    }
    if ((response == (undefined8 *)0x0) || ((tb_message *)*response ==
     ↪  (tb_message *)0x0)) {
      return 4;
    }
    tb_message_set_state((tb_message
     ↪  *)*response,TB_MESSAGE_STATE_RECEIVED);
    set_tb_msg_disposition((tb_message *)*response,2);
    tb_message_tpt_buffer = tb_message_get_transport_buffer((tb_message
     ↪  *)*response);
    if ((*(ushort *)(tb_message_tpt_buffer + 0x2a) & 1) == 0) {
                    /* Failure */
      return 0;
    }
    local_70 = FUN_fffffff0088eeef4((tb_message *)*response);
    if (local_70 == 0) {
      local_70 = FUN_fffffff0082bab90();
      FUN_fffffff0088eef00(*response);
    }
    puVar4 = connection->tb_list;
    if (puVar4 == (undefined8 *)0x0) goto LAB_fffffff0088ede14;
    puVar2 = (undefined8
     ↪  *)_tb_message_accumulator_accumulate(puVar4,*response);
    message_00 = (tb_message
     ↪  *)allocate_in_structure(&DAT_fffffff007c2a670,4);
    if (message_00 != (tb_message *)0x0) {
      tpt_buffer = (tb_bufffer
       ↪  *)allocate_in_structure(&DAT_fffffff007c2a6b0,4);
      if (tpt_buffer != (tb_bufffer *)0x0) {
        uVar5 = __tb_connection_message_construct(connection,0,message_
         ↪  00,tpt_buffer,0,0);
        if ((int)uVar5 != 0) {
          FUN_fffffff00816b9e8(&DAT_fffffff007c2a6f0,tpt_buffer);
          puVar3 = &DAT_fffffff007c2a730;
LAB_fffffff0088edc98:
          FUN_fffffff00816b9e8(puVar3,message_00);
          return uVar5;
        }
        if (puVar2 == (undefined8 *)0x0) {
          while( true ) {
            message_complete(message_00);
            *(ushort *)&tpt_buffer->someFlags = *(ushort
             ↪  *)&tpt_buffer->someFlags | 4;
            tb_message_set_state(message_00,3);
            if (((long *)connection->observers != (long *)0x0) &&
               (tb_message_tpt_buffer = *(long *)connection->observers,
                ↪  tb_message_tpt_buffer != 0))
            {
              (**(code **)(tb_message_tpt_buffer + 0x10))
```

```
                        (tb_message_tpt_buffer + 0x10,tb_message_tpt_bu
                         ↪  ffer,transport,message,
                         response,some_sort_of_state);
            }
            uVar5 = tb_transport_send_message(transport);
            if ((int)uVar5 != 0) {
              __tb_connection_message_destruct(connection);
              FUN_fffffff00816b9e8(&DAT_fffffff007c2a770,tpt_buffer);
              puVar3 = &DAT_fffffff007c2a7b0;
              goto LAB_fffffff0088edc98;
            }
            FUN_fffffff0088eef00(message_00,local_70);
            puVar2 = (undefined8 *)_tb_message_accumulator_accumulate(p
             ↪  uVar4,message_00);
            if (puVar2 != (undefined8 *)0x0) break;
            FUN_fffffff0088ee078(connection,message_00,0,0,0);
          }
        }
        __tb_connection_message_destruct(connection,message_00);
        FUN_fffffff00816b9e8(&DAT_fffffff007c2a7f0,tpt_buffer);
        FUN_fffffff00816b9e8(&DAT_fffffff007c2a830,message_00);
        puVar4 = (undefined8
         ↪  *)tb_message_get_transport_buffer((tb_message *)*response);
        FUN_fffffff0088ee764(transport,puVar4);
        FUN_fffffff0088ee80c(puVar4);
        *puVar4 = *puVar2;
        puVar4[3] = puVar2[3];
        *(undefined1 *)(puVar4 + 5) = 1;
        *(undefined2 *)((long)puVar4 + 0x2a) = *(undefined2
         ↪  *)((long)puVar2 + 0x2a);
        FUN_fffffff00816b9e8(&DAT_fffffff007c2a870,puVar2);
        return 0;
      }
      goto LAB_fffffff0088ede1c;
    }
  }
  else {
    ASSERT_FAIL_QUERY_STATE==TB_MESSAGE_STATE_READY();
LAB_fffffff0088ede10:
    TB_ASSERT_FAIL_QUERY_DISPOSITION==TB_MESSAGE_DISPOSITON_QUERY();
LAB_fffffff0088ede14:
    FUN_fffffff0088ee4b4();
  }
  FUN_fffffff0088ee488();
LAB_fffffff0088ede1c:
  uVar5 = FUN_fffffff0088ee45c();
  if ((*(ulong **)(uVar5 + 8) != (ulong *)0x0) && (uVar5 = **(ulong
   ↪  **)(uVar5 + 8), uVar5 != 0)) {
                    /* WARNING: Could not recover jumptable at
                     ↪  0xfffffff0088ede3c. Too many branches
                        */
                    /* WARNING: Treating indirect jump as call */
    uVar5 = (**(code **)(uVar5 + 0x10))();
    return uVar5;
  }
  return uVar5;
```

```
146 }
```

Listing A.14: Kernel `tb_connection_send_query` function for sending a Tightbeam message.

A.30 lldb Trace on `audiomxd` on Startup with Audio Capture

The breakpoint is set to `exclaves_*`, reduced excerpt.

```
lldb)  bt
  thread #49, queue = 'AudioControl', stop reason = breakpoint 1.18
    frame #0: 0x00000001f18d0c0c libsystem_kernel.dylib`exclaves_sensor_start
    frame #1: 0x00000001b2a4a7a8 MediaExperience`-[MXExclaves
     ↪  updateSensorStatus:reason:]
(lldb)  bt
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', queue =
   ↪  'com.apple.AudioServerDriverTransports.HPMic.serial', stop reason =
   ↪  breakpoint 1.17
    frame #0: 0x00000001f18d0bd8 libsystem_kernel.dylib`exclaves_sensor_create
    frame #1: 0x0000000264839a7c AudioServerDriverTransports_Base'
(lldb)  bt
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', queue =
   ↪  'com.apple.AudioServerDriverTransports.HPMic.serial', stop reason =
   ↪  breakpoint 1.3
    frame #0: 0x00000001f18d0b38
     ↪  libsystem_kernel.dylib`exclaves_audio_buffer_create
    frame #1: 0x000000026484b78c AudioServerDriverTransports_Base
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', queue =
   ↪  'com.apple.AudioServerDriverTransports.HPMic.serial', stop reason =
   ↪  breakpoint 1.19
    frame #0: 0x00000001f18d0c74 libsystem_kernel.dylib`exclaves_sensor_status
    frame #1: 0x0000000264839e5c AudioServerDriverTransports_Base
(lldb)  bt
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', queue =
   ↪  'com.apple.AudioServerDriverTransports.IOA2.device.HPMic.ioReference',
   ↪  stop reason = breakpoint 1.19
    frame #0: 0x00000001f18d0c74 libsystem_kernel.dylib`exclaves_sensor_status
    frame #1: 0x0000000264839e5c AudioServerDriverTransports_Base
(lldb)  bt
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', queue =
   ↪  'com.apple.AudioServerDriverTransports.IOA2.device.HPMic.ioReference',
   ↪  stop reason = breakpoint 1.19
    frame #0: 0x00000001f18d0c74 libsystem_kernel.dylib`exclaves_sensor_status
    frame #1: 0x0000000264839e5c AudioServerDriverTransports_Base`
(lldb)  bt
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', stop reason = breakpoint
   ↪  1.2
    frame #0: 0x00000001f18d0ba0
     ↪  libsystem_kernel.dylib`exclaves_audio_buffer_copyout_with_status
    frame #1: 0x000000026484bb10 AudioServerDriverTransports_Base
(lldb)  bt
  thread #62, name = 'audio IO: VAD [vdef] AggDev 11', stop reason = breakpoint
   ↪  1.2
    frame #0: 0x00000001f18d0ba0
     ↪  libsystem_kernel.dylib`exclaves_audio_buffer_copyout_with_status
    frame #1: 0x000000026484bb10 AudioServerDriverTransports_Base
```

Listing A.15: lldb trace on `audiomxd` on startup with audio capture, breakpoint set to `exclaves_*`, reduced excerpt.

A.31 lldb Trace Excerpt on `exclaves_sensor_start`

Trace excerpt taken from `corespeechd`.

```
[ Legend: Modified register | Code | Heap | Stack | String ]
------------------registers-----------------
```

```
x0     : 0x0
x1     : 0x1d0cb7436     -> ("com.apple.sensors.mic"?)
x2     : 0x10069d340
x3     : 0x16fdb9cf0
x4     : 0x20140b098    (libsystem_trace.dylib 0x1ebb8b098)
 ↪  -><_os_log_current_test_callback+0>
x5     : 0x3b34dcdd02040004
x6     : 0x0
x7     : 0x0
x8     : 0xf52979e45a5f00a3
x9     : 0xf52979e45a5f00a3
x10    : 0x1000001000000000
x11    : 0x64d0
x12    : 0x0
x13    : 0x200000110000e40
x14    : 0x20000001000
x15    : 0x200000010000df0
x16    : 0x1ea190bd8    (libsystem_kernel.dylib 0x1d4910bd8)
 ↪  -><exclaves_sensor_create+0>
x17    : 0x204d4fbc0
x18    : 0x0
x19    : 0x10069d330
x20    : 0x100695b00
x21    : 0x1fcba6000
x22    : 0x20384a000    (CoreSpeechFoundation 0x1edfca000)
 ↪  -><CSAudioFileManager+24>
x23    : 0x0
x24    : 0x0
x25    : 0x0
x26    : 0x0
x27    : 0x0
x28    : 0x0
fp     : 0x16fdbb0d0
lr     : 0x1d0c09234    (CoreSpeechFoundation 0x1bb389234)
 ↪  -><-[CSExclaveIndicatorController init]+180>
sp     : 0x16fdbb070
pc     : 0x1ea190bd8    (libsystem_kernel.dylib 0x1d4910bd8)
 ↪  -><exclaves_sensor_create+0>
cpsr   : [n Z C v q ssbs pan dit ge e a i f m]
-----------------stack-----------------
0x16fdbb070|+0000: 0x0000000000000000  <- $sp
0x16fdbb078|+0008: 0x0000000000000000
0x16fdbb080|+0010: 0x0000000010069d330
0x16fdbb088|+0018: 0x0000000203846f40 (CoreSpeechFoundation 0x1edfc6f40)  ->
 ↪  <CSExclaveIndicatorController+0>
0x16fdbb090|+0020: 0xd0cb741108200102
0x16fdbb098|+0028: 0x0000000000000001
0x16fdbb0a0|+0030: 0x01338001a8e7e978
0x16fdbb0a8|+0038: 0xf52979e45a5f00a3
-----------------code-----------------
libsystem_kernel.dylib`exclaves_sensor_create:
0x1ea190bd8 <+0>:  pacibsp
0x1ea190bdc <+4>:  stp    x29, x30, [sp, #-0x10]!
0x1ea190be0 <+8>:  mov    x29, sp
0x1ea190be4 <+12>: mov    x3, x2
0x1ea190be8 <+16>: mov    x2, x1
0x1ea190bec <+20>: mov    w1, #0xa000000 ; =167772160
0x1ea190bf0 <+24>: mov    x4, #0x0 ; =0
0x1ea190bf4 <+28>: mov    x5, #0x0 ; =0
0x1ea190bf8 <+32>: mov    x6, #0x0 ; =0
-----------------threads-----------------
thread #1: tid = 0x64d0, 0x00000001ea190bd8
 ↪  libsystem_kernel.dylib`exclaves_sensor_create, queue =
 ↪  'com.apple.main-thread', stop reason = breakpoint 1.21
thread #2: tid = 0x64d1, 0x00000001ea179a90
 ↪  libsystem_kernel.dylib`__workq_kernreturn + 8
thread #3: tid = 0x64d3, 0x00000002234e4e68 libxpc.dylib`xpc_int64_get_value,
 ↪  queue = 'com.apple.xpc.activity.com.apple.cs.postinstall'
```

```
-----------------trace-----------------
[#0] 0x1ea190bd8 (libsystem_kernel.dylib 0x1d4910bd8)   ->
↪  exclaves_sensor_create()
[#1] 0x1d0c09234 (CoreSpeechFoundation 0x1bb389234)   ->
↪  -[CSExclaveIndicatorController init]()
[#2] 0x1d0c28e50 (CoreSpeechFoundation 0x1bb3a8e50)   -> -[CSExclaveRecordClient
↪  init]()
[#3] 0x1d0c28f68 (CoreSpeechFoundation 0x1bb3a8f68)   ->
↪  __37+[CSExclaveRecordClient sharedClient]_block_invoke()
[#4] 0x1a049f584 (libdispatch.dylib 0x18ac1f584)   -> _dispatch_client_callout()
[#5] 0x1a04889a8 (libdispatch.dylib 0x18ac089a8)   -> _dispatch_once_callout()
[#6] 0x1d0c28f44 (CoreSpeechFoundation 0x1bb3a8f44)   -> +[CSExclaveRecordClient
↪  sharedClient]()
[#7] 0x1d0c2ca04 (CoreSpeechFoundation 0x1bb3aca04)   ->
↪  +[CSExclaveMessageHandlingFactory commonExclaveMessageHandler]()
[#8] 0x1d0c25d18 (CoreSpeechFoundation 0x1bb3a5d18)   ->
↪  -[CSExclaveAssetManagerProxy init]()
[#9] 0x1d0c25dc0 (CoreSpeechFoundation 0x1bb3a5dc0)   ->
↪  __43+[CSExclaveAssetManagerProxy sharedManager]_block_invoke()
[#10] 0x1a049f584 (libdispatch.dylib 0x18ac1f584)   ->
↪  _dispatch_client_callout()
[#11] 0x1a04889a8 (libdispatch.dylib 0x18ac089a8)   -> _dispatch_once_callout()
[#12] 0x1d0c25d9c (CoreSpeechFoundation 0x1bb3a5d9c)   ->
↪  +[CSExclaveAssetManagerProxy sharedManager]()
[#13] 0x1d0c80520 (CoreSpeechFoundation 0x1bb400520)   ->
↪  -[CSUAFAssetManagerBase init]()
[#14] 0x1d0c80408 (CoreSpeechFoundation 0x1bb400408)   ->
↪  -[CSUAFAssetManagerBase initWithForceSetIsExclave:exclaveManagerProxy:]()
[#15] 0x1d0c4520c (CoreSpeechFoundation 0x1bb3c520c)   ->
↪  __35+[CSUAFAssetManager sharedInstance]_block_invoke()
[#16] 0x1a049f584 (libdispatch.dylib 0x18ac1f584)   ->
↪  _dispatch_client_callout()
[#17] 0x1a04889a8 (libdispatch.dylib 0x18ac089a8)   -> _dispatch_once_callout()
[#18] 0x1d0c451cc (CoreSpeechFoundation 0x1bb3c51cc)   -> +[CSUAFAssetManager
↪  sharedInstance]()
[#19] 0x1000957f4 (corespeechd 0x1001517f4)   -> ___lldb_unnamed_symbol3044()
```

Listing A.16: lldb trace excerpt on `exclaves_sensor_start` from `corespeechd`.