

From Fault to Oracle: Breaking Google Pixel TPU Hardware Memory Isolation via Trusted Deserialization Faults

Minghao Lin
mlin7282@usc.edu
University of Southern California
Los Angeles, California, USA

Guanxing Wen
guanxing.wen@certik.com
Certified Kernel Tech LLC
New York, New York, USA

Haoxuan Xu
xuhaoxua@usc.edu
University of Southern California
Los Angeles, California, USA

Yuzhou Fang
yuzhou.fang@usc.edu
University of Southern California
Los Angeles, California, USA

Nanzi Yang
nyang@odu.edu
Old Dominion University
Norfolk, Virginia, USA

Xiaokuan Zhang
xiaokuan@gmu.edu
George Mason University
Fairfax, Virginia, USA

Yanan Guo
yanan.guo@rochester.edu
University of Rochester
Rochester, New York, USA

Zhuo Zhang
zz@cs.columbia.edu
Columbia University
New York, New York, USA

Mengyuan Li
mli49061@usc.edu
University of Southern California
Los Angeles, California, USA

Abstract

Modern mobile systems increasingly rely on dedicated AI/ML accelerators, such as neural processing units (NPUs) and Tensor Processing Units (TPUs), to meet the latency, energy, and privacy demands of on-device inference. These accelerators are often treated as part of the trusted hardware base (THB), with their code and memory protected by surrounding trusted components. On Google's Pixel platform, the TPU is intended to remain inaccessible even to a compromised host, with protection anchored by the Google Tensor Security Core (GSC). In this paper, we analyze the isolation architecture of the Pixel TPU and the protection boundary enforced by GSC. We show that, although the host cannot directly cross the hardware isolation boundary, it can still break the intended confidentiality guarantee by using two legitimate channels: the kernel control interface (KCI) and the fault-log buffer. This composition yields a fault-oracle attack that enables arbitrary reads from protected TPU memory, with fault metadata exposed through the log interface serving as a repeatable disclosure channel. The underlying issue is a structural disconnect in GSC's boundary enforcement. While GSC validates the top-level host-supplied buffer at the isolation boundary, the zero-copy parser could reinterpret attacker-controlled offsets as trusted derived pointers, without re-establishing the original memory-safety invariant.

We further build a lightweight static taint analyzer and systematically identify fault-oracle gadgets across command handlers. On the two Google Pixel A-series production devices, the Pixel 9a and the Pixel 10a, we found 21 reachable sites, through which we could recover the complete protected TPU image, including both the protected DRAM contents and the on-chip runtime state. We responsibly disclosed the vulnerabilities to Google through the Android Vulnerability Reward Program, and Google acknowledged the report and initiated remediation.

CCS Concepts

• **Security and privacy → Hardware attacks and countermeasures**.

Keywords

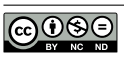
Android, TPU Hardware Memory Isolation, Fault-Oracle Attack

ACM Reference Format:
Minghao Lin, Guanxing Wen, Haoxuan Xu, Yuzhou Fang, Nanzi Yang, Xiaokuan Zhang, Yanan Guo, Zhuo Zhang, and Mengyuan Li. 2026. From Fault to Oracle: Breaking Google Pixel TPU Hardware Memory Isolation via Trusted Deserialization Faults. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846589

1 Introduction

On-device machine learning has become a core part of modern mobile platforms. To meet latency, energy, and privacy requirements, modern phones increasingly offload inference to dedicated ML accelerators, such as neural processing units (NPUs) and Tensor Processing Units (TPUs), that operate close to cameras, microphones, and user-facing applications. These accelerators are no longer passive arithmetic units hidden behind a driver: their stacks include device-local firmware that parses host commands, manages shared buffers, schedules accelerator work, and records faults. This creates a difficult system security question: *where should the security boundary be drawn in a heterogeneous mobile platform, and how is that boundary actually enforced?*

In this paper, we systematically demystify how Google protects its on-device TPU. Beginning with the Tensor SoC, Google introduced the Google Tensor Security Core (GSC), a dedicated security subsystem described as physically and logically separate from the application processor and as extending protection beyond the CPU-side TrustZone environment [2, 10, 21, 22]. Android documentation



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846589

further uses the term *trusted hardware base* (THB) for security-sensitive hardware components outside the rich OS, and identifies the EdgeTPU as part of this THB for on-device ML functionality [1].

The intended security goal of this design is straightforward: even if the main processor or Android kernel is compromised, the privileged and malicious host should not be allowed to inspect, replace, or tamper with the TPU's protected code and runtime state. Otherwise, an attacker could recover secrets, sensitive model data, firmware metadata, heap and stack contents, or infer internal TPU design details. However, Google does not publicly document how this protection is actually enforced. The concrete architecture of EdgeTPU protection, including its isolation boundary, trusted components, exposed host interfaces, and residual attack surface, remains unclear.

To understand how this boundary is enforced, we analyze the TPU protection architecture from the host side outward. We start from artifacts and behavior visible before or outside the GSC-protected runtime: the Android open-sourced EdgeTPU driver, the firmware-loading path exposed through the host-GSC interface, the staged TPU firmware image before it is locked, and the shared-memory regions that remain visible after initialization. Together, these observations let us reconstruct how the GSC enforces the protection boundary through a load-then-lock sequence: GSC first authenticates the TPU firmware and then revokes CPU access to the protected staging region; the TPU's embedded microcontroller unit (MCU) subsequently relocates performance-critical firmware components into on-chip memory. Under this design, even a fully compromised Android kernel cannot remap or access the protected firmware region, replace the authenticated image, or inspect the live TPU runtime.

At the same time, protection does not eliminate all host-visible interaction. After initialization, two architecturally sanctioned channels remain exposed to the untrusted host: the kernel control interface (KCI), through which Android submits TPU workloads, and a fault-log buffer, through which firmware reports diagnostics. Both channels are deliberately designed as mediated interfaces that support necessary host-TPU coordination without granting the untrusted host access to protected TPU memory.

We show that the remaining mediated interfaces are sufficient to defeat the intended protection boundary. Our key insight is that a host-visible fault path, although designed only for diagnostics, can be turned into an oracle for probing protected TPU memory. We develop ***Fault-Oracle***, an attack that extracts protected state even though the underlying memory remains behind a hardware-enforced boundary and is never directly mapped to the host.

At a high level, Fault-Oracle works by inducing trusted firmware to dereference attacker-chosen protected addresses and then using exported fault metadata to infer the outcome of those protected accesses. The root cause is a mismatch between where the hardware boundary is enforced and where trusted parsing actually occurs. At the boundary, GSC prevents the host from directly mapping or accessing protected TPU memory, but it does not automatically ensure that every pointer later derived by trusted TPU firmware still stays within the host-visible region. In the EdgeTPU firmware, the boundary check validates the top-level command buffer, while the trusted zero-copy parser derives nested pointers from attacker-controlled offsets without revalidating the resulting addresses. As a result, crafted KCI requests can redirect trusted parsing logic from the validated command buffer into protected TPU memory. When the resulting access triggers a firmware fault, the fault-log interface reports enough structured metadata to make the fault repeatable and interpretable. We refer to this flaw pattern as *trusted deserialization faults*. To the best of our knowledge, *Fault-Oracle* is the first attack to turn architecturally permitted host-TPU interfaces into a persistent arbitrary-read channel across Google's hardware-enforced EdgeTPU protection boundary.

This framing also suggests that the issue may not be limited to the first gadget we found. If the interface checks only the outer command buffer, then any parser code that derives trusted pointers from host-controlled serialized fields can recreate the same boundary violation. We therefore build a lightweight scanner for this semantic pattern: a host-controlled field is used to derive a trusted pointer, and the derived pointer is later dereferenced without a boundary check. This rule captures the main implementation forms we observe in the firmware and filters out unrelated length checks and ordinary field validation. The scan shows that the weakness is widespread. Across the five inner command handlers, we find 23 Fault-Oracle sites, 21 of which are reachable on both devices evaluated: the Pixel 9a and Pixel 10a, the two Pixel A-series phones powered by the Google Tensor G4 SoC; each reported site is manually validated. Using the primitive, we dump the protected DRAM firmware region with 99.03 % verified accuracy against ground truth, or 99.97 % after modeling loader-time relocations. We also recover the complete 57 KiB on-chip executable with 100 % coverage, with all disassembly-based fidelity metrics above 96 %. Together, these results show that unsafe zero-copy deserialization is a systemic weakness in the EdgeTPU, not a one-off bug.

In summary, this paper makes the following contributions:

- We reconstruct the undocumented Google EdgeTPU protection architecture, including firmware authentication, SoC-enforced memory isolation, runtime relocation, and the host-visible interfaces that remain after initialization.
- We present *Fault-Oracle*, the first attack that turns architecturally permitted host-TPU interfaces into a persistent arbitrary-read channel across Google's hardware-enforced EdgeTPU protection boundary.
- We identify *trusted deserialization faults* as the root cause: trusted firmware derives pointers from host-controlled serialized data and dereferences them without rechecking the intended memory boundary.
- We show that the weakness is systemic, finding 23 Fault-Oracle sites across command handlers and using them to recover protected DRAM firmware with 99.03 % exact-match accuracy (99.97 % after attributing loader-time relocations) and the complete 57 KiB on-chip executable with 100 % coverage. The primitive can achieve an end-to-end disclosure throughput of ≈ 233 B/s over the standard KCI submission interface.

Responsible disclosure. We responsibly disclosed the Fault-Oracle primitive, all scanner-identified sites, and the resulting protected-memory disclosure impact to Google through the Android Vulnerability Reward Program (VRP) in March 2026. Google acknowledged the report and subsequently awarded a bounty under the

VRP. Google has also confirmed that a CVE will be assigned to the reported issue. Google has also since developed and deployed remediation addressing the reported issues.

2 Background

2.1 Mobile ML Accelerators in the Trusted Hardware Base

On mobile platforms, on-device ML accelerators are part of a broader shift toward host-driven heterogeneous systems: the CPU runs the rich OS and applications, while specialized processors execute workloads that must meet tighter performance, energy, and latency constraints. This trend has become particularly prominent with the rise of edge AI [23, 29], where resource-constrained devices must support increasingly demanding machine-learning workloads under strict power, thermal, and latency budgets. Modern platforms therefore integrate multiple specialized processing units rather than relying on the CPU alone.

In these systems, the CPU typically remains the system-wide control point: it runs the operating system, manages applications, and dispatches work to specialized accelerators [34]. The accelerators then execute tasks that are inefficient on the CPU, such as massively parallel graphics and numerical computation on GPUs, neural-network inference on TPUs or NPUs, and isolated security-sensitive functionality on dedicated security cores. This division of labor yields a CPU-driven heterogeneous architecture, in which the CPU remains the global control point while domain-specific cores execute specialized workloads more efficiently.

To further improve efficiency, many modern accelerators incorporate local microcontrollers (MCUs) or firmware-managed control cores [3, 19]. These co-processors often take over functions that would otherwise be handled by the CPU and its host-side driver, including command handling, scheduling, synchronization, memory and buffer management, and device-specific runtime control. Consequently, the host-side driver stack often becomes thinner, while more control logic migrates into privileged device-local firmware [4]. This evolution improves performance, but it also shifts security-critical functionality away from the CPU and into accelerator-resident firmware.

Android's security taxonomy refers to such security-relevant SoC components as part of the *trusted hardware base* (THB). In its vulnerability-severity guidance, Android defines the THB as discrete hardware components, generally on the SoC, that provide functionality critical to the device's core use cases, listing cellular basebands, DSPs, GPUs, and ML accelerators as examples [1]. Thus, an ML accelerator can be security-critical even when it is not a conventional CPU-side trusted execution environment. Its firmware, runtime state, and host-visible interfaces therefore become part of the security boundary that the platform must preserve.

2.2 Google Pixel EdgeTPU and GSC Protection

Beginning with the Pixel 6 series, Google adopted its in-house Google Tensor SoC in Pixel phones to improve on-device machine-learning performance [10, 21]. A key component of this platform is the EdgeTPU, a dedicated ML accelerator integrated into the Google Tensor SoC [10]. The EdgeTPU executes latency- and energy-sensitive on-device ML workloads, allowing Pixel devices to run inference close to mobile applications and sensor-derived inputs rather than relying only on the CPU or cloud services. Prior to its deployment in Tensor-based Pixel devices, EdgeTPU had already appeared in other Google products, including Coral accelerators [5] and earlier Pixel platforms [30].

Google also introduced the Google Tensor Security Core (GSC) as a dedicated hardware security subsystem for protecting sensitive on-device components [22]. Google describes TrustZone as a key part of its security architecture for general secure processing, but also emphasizes that the security improvements in Google Tensor go beyond TrustZone. In particular, GSC is physically and logically distinct from the application processor and serves as the protection anchor for security-sensitive components such as the EdgeTPU [22]. Under Android's THB taxonomy, the Pixel EdgeTPU is an ML processor that provides functionality critical to on-device ML use cases [1]. This places the EdgeTPU's trust boundary outside the conventional CPU-side TrustZone environment: the Android host may submit work through intended interfaces, while protected TPU firmware and runtime state should remain inaccessible to the rich OS. However, the detailed security architecture by which GSC protects the EdgeTPU remains undocumented, leaving the practical protection boundary and attack surface unclear.

3 Demystify EdgeTPU Hardware-Isolation Design and Residual Channels

In this section, we reconstruct the EdgeTPU hardware-isolation design, focusing on both the post-initialization protection boundary and the communication surface that remains across it. We first show how GSC verification, firewall locking, and TPU-MCU relocation create CPU-inaccessible protected regions in the TPU virtual address space. We then demystify the two host-visible channels that remain between the CPU and the protected TPU runtime: the log ringbuffer and the KCI command/response interface. We use *channel* to mean a protocol endpoint consisting of a host-visible DRAM region, a synchronization mechanism, and TPU-side firmware logic that consumes or produces records in that region. By reconstructing the TPU-side logging, dispatch, and zero-copy parsing logic behind these channels, we identify the residual cross-boundary behavior that our attack later exploits in §4.

3.1 Post-initialization Isolation Boundary

We first characterize the EdgeTPU post-initialization isolation boundary through a combination of host-visible artifacts, firmware-loading behavior, and binary analysis. The boundary is established by a secure loading sequence that combines GSC-enforced firmware verification and memory locking with TPU-MCU–driven relocation of performance-critical code into on-chip memory. As illustrated in Figure 1, the non-secure host CPU first retrieves the TPU MCU firmware from a fixed host-side path and stages it in DRAM at 0x92e00000–0x92f20000. The host then notifies GSC of the staged region through the mailbox MMIO interface.

Inspection of the extracted GSC image shows that GSC runs Trusty OS and includes trusted applications responsible for EdgeTPU firmware loading. During this process, GSC validates the firmware version and verifies its signature using host-supplied

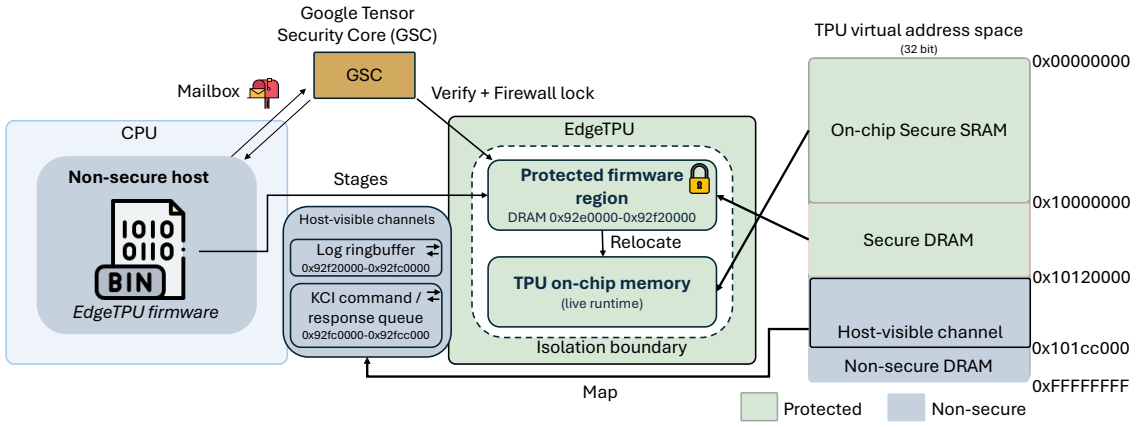


Figure 1: EdgeTPU isolation boundary after initialization. GSC authenticates and firewalls the firmware staging region, leaving the log ringbuffer and KCI command/response queues as DRAM-backed channels to the protected TPU runtime. Left-side ranges denote host physical DRAM addresses; the rightmost column shows the TPU virtual address space, including secure SRAM below 0x10000000, secure DRAM at 0x10000000–0x10120000, and host-visible non-secure mappings above it.

metadata, thereby preventing previously identified MCU-image substitution attacks [19]. After successful validation, GSC programs the SoC firewall to lock the staged firmware DRAM region and revoke subsequent CPU access to that region.

After the staged DRAM region is locked, the TPU MCU relocates performance-critical firmware components into on-chip memory and continues execution from the relocated image. This design yields a distinct TPU-side virtual memory layout: relocated firmware components and live runtime state reside in secure on-chip SRAM below 0x10000000 in the TPU's 32-bit virtual address space, while the authenticated firmware image in staged DRAM is mapped into the secure-DRAM range 0x10000000–0x10120000. Consequently, the live TPU runtime is not mirrored in any host-visible DRAM region after initialization.

These mechanisms define a two-region isolation boundary after initialization. The first protected region is the staged firmware DRAM range 0x92e00000–0x92f20000, which contains the authenticated TPU firmware image and corresponds to the TPU secure-DRAM mapping at 0x10000000–0x10120000. The second protected region is TPU on-chip memory, which stores relocated firmware components and live runtime state below 0x10000000 in the TPU virtual address space. Under this security design, both regions are protected by the SoC firewall and are intended to remain inaccessible to the CPU; only TPU virtual addresses above 0x10120000 map to non-secure DRAM visible to the host.

3.2 Residual Host-visible Channel Model

The isolation boundary described above does not completely remove CPU–TPU communication. This is expected: the EdgeTPU is a host-driven heterogeneous accelerator, and the non-secure CPU must still be able to submit workloads, receive completion status, and collect diagnostic information. Therefore, after initialization, the design reduces the CPU–TPU interface to a small number of residual channels located in non-secure DRAM.

Channel	Host-visible state	Protected TPU logic
Log ringbuffer TPU → host	Non-secure DRAM ringbuffer containing diagnostic and fault records.	Firmware logging routine and exception-time fault handler emit records into the ringbuffer.
KCI queue Host ↔ TPU	Shared command and response queue; MMIO doorbell for workload submission and completion status.	KCI dispatcher invokes command-specific zero-copy parsers over host-supplied buffers.

Table 1: Residual host-visible channels after EdgeTPU initialization. Each channel combines non-secure shared state with protected TPU firmware logic that interprets it.

We use the term *channel* to describe this residual interface because each exposed region has protocol semantics beyond raw shared memory. A channel consists of three elements. First, it has a host-visible DRAM region that both sides agree to interpret according to a specific format. Second, it has a synchronization mechanism, such as producer–consumer ringbuffer state or an MMIO doorbell. Third, it has a TPU-side firmware endpoint, such as a logging routine, exception handler, KCI dispatcher, or command handler, that interprets the shared region while executing inside the protected runtime. Under this model, our analysis identifies two DRAM-backed residual channels relevant to protected-runtime communication: a log channel and a KCI command channel.

3.3 Diagnostic Log Channel

The first residual channel is the log ringbuffer at DRAM 0x92f20000–0x92fc0000. This channel is an outbound diagnostic interface from the protected TPU runtime to the non-secure host. Its normal purpose is to allow the TPU firmware to report progress, warnings,

and failure information without exposing the protected firmware image or live runtime memory to the CPU.

The channel is implemented as a producer–consumer ringbuffer in non-secure DRAM. The TPU firmware acts as the producer, while the non-secure host acts as the consumer. Our testing shows that the firmware embeds numerous developer-inserted logging calls along normal execution paths. When execution reaches one of these sites, the firmware invokes a shared logging routine that formats a diagnostic record and appends it to the ringbuffer. These normal records include a timestamp together with the corresponding diagnostic string. Because the ringbuffer resides outside the firewall-protected firmware region, the host can continue to parse these records after initialization.

The same channel is also used by the firmware's exception-handling path. We observe that the TPU-side runtime enforces memory-access constraints through MPU and IOMMU mechanisms, with different classes of accesses checked by these mechanisms. For example, when the firmware triggers a memory fault by violating access permissions or dereferencing an unmapped address, control enters a dedicated exception handler. This handler writes substantially richer fault records into the same log ringbuffer, including the stack trace, MPU configuration table, and the faulting address reported through Data Fault Address Register (DFAR). Thus, although the log channel is not the primary input channel, it exposes protected-runtime behavior across the isolation boundary. In later sections, we use this channel as a diagnostic oracle to observe whether crafted inputs reach specific fault conditions inside the TPU runtime.

3.4 KCI Command Channel

The second residual channel is the KCI command and response queue at DRAM 0x92fc0000–0x92fcc000. Unlike the log channel, which is primarily produced by the TPU, the KCI channel is initiated by the host. It serves as the host-mediated command path for submitting workloads and control operations to the TPU and for receiving completion status and return values. This interface is architecturally indispensable because the EdgeTPU has no independent workload source: application-supplied data must reach the accelerator through a CPU-mediated submission path, as in other CPU-plus-accelerator systems [34].

KCI transport. At the transport layer, the non-secure host wraps each TPU operation as a KCI request and deposits the request into the shared command queue. The request carries the information needed by the TPU firmware to identify and locate the command payload, including a command selector, a top-level command-buffer pointer, and a declared command-buffer size. The host then signals the TPU through an MMIO doorbell. On the TPU side, a KCI dispatcher receives the request, selects the corresponding command handler, and eventually writes the completion status or return values into the response queue. The host then consumes the response from the same shared region.

This design means that the KCI queue itself does not necessarily contain the full command object. Instead, it acts as a descriptor channel. The complex command payload resides in a separate host-supplied command buffer in non-secure DRAM, and the KCI request tells the TPU where that buffer is and how large it is. Consequently, the TPU firmware crosses the isolation boundary twice during command processing: first when it reads the KCI request from the shared queue, and second when it dereferences the request-provided pointer to parse the command buffer.

Recovering TPU-side handlers. Understanding the TPU-side processing logic is non-trivial because firmware relocation substantially complicates analysis. During initialization, the TPU MCU relocates key routines from DRAM into on-chip memory, so the static firmware image extracted from the host side no longer directly reflects the code layout that executes at runtime. Instructions whose targets depend on runtime placement, including PC-relative branches, `BL` calls, and switch-case dispatches, therefore cannot be resolved correctly from the extracted image alone. This disrupts recovery of both control flow and data flow, causing whole-function decompilation to fail and making disassembly around relocated instructions unreliable.

To recover the relevant logic, we identify KCI command handlers through embedded debug strings that remain present in the firmware image and then manually trace the dataflow of the register holding the top-level command-buffer pointer. A key observation is that, although relocation breaks recovery of PC-relative control flow, the command-buffer parsing logic itself is primarily expressed through local data-movement and arithmetic instructions whose semantics do not depend on PC-relative addressing. By focusing on these relocation-insensitive instructions, we are still able to reconstruct how the handlers process host-controlled input.

Zero-copy command-buffer parsing. Our results show that KCI command handlers parse command buffers using a zero-copy deserialization strategy. The handler first retrieves the top-level command-buffer pointer and declared size from the KCI request. It then validates the top-level buffer before parsing it. Specifically, the handler enforces a *32-bit lower-bound check* on the command-buffer pointer: the buffer must reside at or above 0x11000000 in the TPU virtual address space. This bound is stricter than the general non-secure boundary at 0x10120000. In our recovered layout, the lower non-secure mappings are used for firmware-managed shared metadata such as residual channel state, whereas host-supplied command payloads are expected to reside in the higher command-buffer range. By construction, the 0x11000000 lower bound excludes on-chip SRAM, the secure firmware-mapped region, and lower shared metadata mappings. The firmware also rejects buffers whose declared size, when added to the base pointer, overflows the 32-bit address space. These checks are intended to confine the top-level command buffer to an accepted non-secure command-buffer region.

After these checks succeed, the handler parses the command buffer in place. It does not copy the full command object into protected SRAM before deserialization. Instead, it resolves nested objects directly from the host-supplied buffer. At each parsing layer, the handler reads a relative offset from the current object, adds that offset to the base address of the current object, and treats the resulting address as the next nested object to parse. The same procedure is then repeated across subsequent layers until the full nested structure has been traversed. This zero-copy strategy avoids additional copying overhead, but it also means that protected firmware repeatedly interprets host-controlled DRAM while executing inside the isolation boundary.

Together, these two channels form the primary software interface across the isolation boundary and play complementary roles in the attack developed in §4. The KCI channel is the active input path that lets the host supply structured descriptors and command buffers to be interpreted by protected firmware. The log channel is the passive observation path that exports exception-time state produced by that firmware back into host-visible DRAM. The attack in §4 therefore does not require direct CPU access to secure SRAM or secure DRAM; it operates entirely through the residual channel semantics reconstructed in this section.

4 Fault-Oracle Attack

In this section, we present the Fault-Oracle attack. Our attack composes two remaining host-visible channels, each preserved for a legitimate purpose, to violate the intended hardware isolation through software-triggered faults. This composition is security-critical because it gives the attacker visibility into protected TPU memory, effectively turning the isolated runtime into a debuggable target from the host side and substantially lowering the barrier to stable follow-on exploitation. We begin by defining the threat model, which clarifies the attacker's capabilities and the constraints under which the attack operates. We then present the Fault-Oracle primitive, showing how attacker-controlled input delivered through the KCI command path can steer trusted parsing logic into protected memory, and how fault metadata returned through the fault-log path turns that fault into a disclosure channel. Finally, we demonstrate the attack through a stack-cookie recovery case study, which concretely illustrates both the workflow and the practical impact of the attack.

4.1 Threat Model

We assume an adversary that has compromised the Pixel Android kernel on the CPU side. Due to THB protection, the privileged adversary can interact with the EdgeTPU only through the standard CPU-to-TPU interface by submitting crafted requests and shared-memory buffers. The attacker cannot achieve code execution inside the trusted EdgeTPU domain and cannot directly read or remap hardware-protected EdgeTPU memory since the EdgeTPU memory is hardware-protected by Google GSC.

We explicitly exclude physical attacks, side channels, hardware fault injection, firmware rehosting, and arbitrary replacement of the EdgeTPU firmware. Although the firmware image can be obtained from the host side for offline analysis, the trusted firmware authenticated by GSC and loaded at runtime cannot be modified by the attacker. This threat model reflects the realistic constraints faced by a software adversary on commodity Pixel devices: compromise of the rich OS does not imply compromise of the isolated EdgeTPU runtime, whose protected memory remains inaccessible even to a hostile kernel. The attacker must therefore operate entirely through the remaining architecturally sanctioned CPU-to-TPU interaction surface.

Under this model, we show that these seemingly narrow and legitimate interfaces are nevertheless sufficient to disclose protected EdgeTPU contents, thereby violating the intended hardware isolation boundary.

4.2 Fault-Oracle Primitive

We now present the core attack primitive that turns two individually legitimate host-visible channels into a protected-memory disclosure oracle. The key insight is that the EdgeTPU enforces the isolation boundary only at the entry point of command parsing, but later treats attacker-influenced, recursively derived pointers as trusted firmware pointers. By combining this unchecked pointer derivation with host-visible fault diagnostics, an attacker can convert failed protected-memory dereferences into concrete memory values.

Insight: validation stops at the parsing entry point. The KCI command path does perform lower-bound and overflow checks, but these checks apply only once, to the top-level command-buffer pointer supplied by the host. After this initial validation, EdgeTPU parses nested command objects in a zero-copy manner. For each nested field, the handler reads a 32-bit relative offset from the command buffer and adds it to a running pointer to derive the base address of the next object. The flaw is that these derived pointers are not revalidated against the post-initialization isolation boundary. Since the arithmetic is performed in unsigned 32-bit arithmetic, an attacker can choose a large offset that wraps the derived pointer across the address-space boundary and redirects it into protected regions, including secure on-chip SRAM and the firewalled firmware image. In effect, EdgeTPU validates the original parsing root, but not the pointer chain produced during parsing.

This flaw is structural rather than a single missing check. The same offset-resolution idiom appears repeatedly in EdgeTPU's zero-copy deserialization logic. Different command handlers implement variants based on addition, subtraction, field-offset addition, and stride-based iteration, but they share the same security mistake: attacker-controlled offsets are promoted into trusted firmware pointers without a second boundary check. We catalog these variants in §5 (so called Fault-Oracle Gadgets).

Composing two benign channels into a fault oracle. The fault oracle attack emerges by combining attacker-controlled KCI inputs with host-visible fault diagnostics. The KCI queue gives the host control over the serialized command payload consumed by trusted TPU firmware. The unchecked zero-copy derivation then turns fields inside this payload into trusted pointers. Separately, the fault-log buffer is designed to expose diagnostic information when firmware dereferences an invalid address. In isolation, each channel appears benign: one submits work, and the other reports faults. Together, however, they create a feedback loop from attacker-controlled command bytes to observable fault addresses. We call this feedback loop the *Fault-Oracle primitive*.

From a wrapped pointer to a leaked word. We illustrate the primitive using the subtraction variant used in our case study. Let P_1 be the validated top-level command-buffer pointer, and let x be an attacker-controlled field offset inside that buffer. During nested parsing, the firmware derives a secondary pointer:

$$P_2 = P_1 + *[P_1 + x].$$

Because P_1 is known and $*[P_1 + x]$ is attacker-controlled, the attacker can choose the offset so that the 32-bit addition wraps and lands at an arbitrary protected TPU address. The firmware then dereferences P_2 and reuses the loaded 32-bit word in a subsequent

address computation:

$$P_3 = P_2 - *P_2.$$

At this point, the attacker controls the first-stage target P_2, but not the second-stage address P_3, because P_3 depends on the unknown protected word stored at P_2. This dependency is precisely what turns the fault into an oracle. When the dereference at P_3 faults, the hardware records P_3 in the data fault address register (DFAR), and the firmware exception handler exports that value through the host-readable fault-log buffer. Since P_2 is attacker-chosen and DFAR reveals P_3, the protected word is recovered as:

$$*P_2 = P_2 - \text{DFAR}.$$

From a single disclosure to reliable arbitrary read. The primitive is not a one-shot crash oracle. The firmware treats the relevant fault as a recoverable warning: it emits the diagnostic record, returns control to the runtime, and continues processing later KCI requests. Each query therefore leaks one attacker-selected 32-bit word from protected TPU memory. By issuing queries over successive target addresses, the attacker upgrades a 4-byte disclosure into a persistent arbitrary-read primitive over protected firmware and runtime memory.

Two properties make the primitive reliable in practice. First, most of the TPU 32-bit virtual address space is unmapped: the active protected memory footprint occupies only the region below 0x10120000. As a result, most attacker-induced second-stage addresses P_3 fault deterministically. Second, even when P_3 falls into a mapped region, the value-dependent computation frequently produces misaligned addresses. Since many firmware loads require alignment, misalignment alone is sufficient to trigger a fault and expose DFAR through the log channel.

Sliding-offset fallback for broader coverage. A practical challenge is that the oracle does not always fire at the exact target address. If the value loaded from P_2 happens to produce a second-stage address P_3 that is mapped and sufficiently well aligned, the access may complete without generating a diagnostic fault. A naive oracle would therefore miss such words and leave holes in the recovered memory image. This is especially problematic when reading contiguous protected regions, where even a small number of non-faulting words can break downstream parsing of firmware structures, pointer tables, or runtime metadata.

To close these gaps, we introduce a sliding-offset fallback that turns nearby faulting queries into overlapping byte windows over the same target word. Rather than requiring the aligned query at P_2 to fault, the attacker additionally queries shifted addresses $P_2 + \delta$ for $\delta \in \{\pm1, \pm2, \pm3\}$. Each shifted query leaks a 4-byte window that partially overlaps the original target word: positive shifts reveal suffix bytes, while negative shifts reveal prefix bytes. By combining complementary windows, such as $\{P_2 - 1, P_2 + 2\}$ or $\{P_2 - 2, P_2 + 1\}$, the attacker reconstructs all four bytes of the original word even when the direct query fails. This fallback is a key part of our exploit design: it converts an address-dependent, partially available oracle into a robust memory-recovery primitive that can cover substantially more of the protected address space.

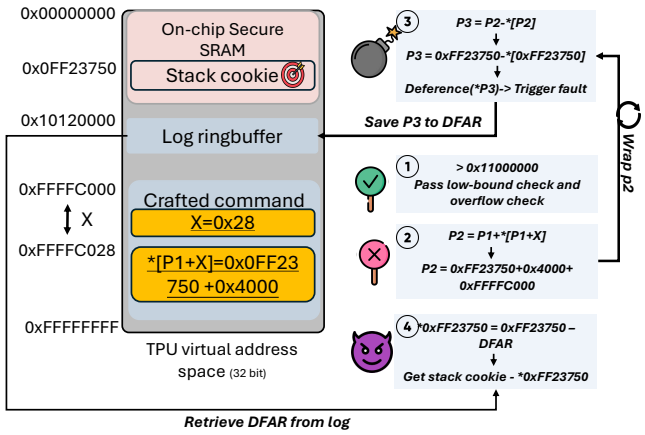


Figure 2: End-to-end Fault-Oracle attack recovering the TPU stack cookie from protected on-chip SRAM. Concrete parameter values are annotated in the figure.

4.3 Case Study: Stack-Cookie Recovery

To illustrate the attack primitive concretely, we present a case study that recovers the TPU stack cookie, a security-critical runtime secret whose disclosure enables subsequent memory-corruption exploitation. This value is neither accessible nor predictable from the CPU side: it is generated at TPU boot time using a SplitMix64 PRNG seeded from the Arm Generic Timer (`CNTPCT_EL0`), stored at `0x0FF23750` in protected on-chip SRAM, and checked by firmware functions throughout execution. Because the cookie resides in protected on-chip SRAM, the CPU-side host cannot directly read it. Once leaked, however, it can be reused within the same TPU session to bypass stack-protector checks in subsequent memory-corruption attacks.

Figure 2 illustrates the concrete attack process. ❶ The attacker crafts a KCI command with base pointer P_1 = `0xFFFFC000`, which passes the firmware's 32-bit unsigned lower-bound check since $P_1 \geq$ `0x11000000` and satisfies the accompanying base-plus-size overflow check. ❷ The command buffer contains the field offset x = `0x28` and an attacker-controlled value $*[P_1{+}x]$ = `0x0FF23750` + `0x4000`, chosen so that the derived pointer $P_2 = (P_1 + *[P_1{+}x]) \bmod 2^{32}$ wraps past the 32-bit boundary to `0x0FF23750`, the address of the stack cookie in protected on-chip SRAM. ❸ The firmware dereferences P_2 to load the live cookie value and then performs the second-stage dereference at $P_3 = (P_2 - *P_2) \bmod 2^{32}$, which triggers a fault because the resulting address either falls outside any mapped region or is misaligned. The hardware captures P_3 in the DFAR, and the firmware's exception handler writes this value into the host-readable fault log. ❹ The host reads the log and recovers the cookie using the unsigned 32-bit subtraction $*$`0x0FF23750` = (`0x0FF23750` − DFAR) mod 2^{32}.

To confirm that the primitive yields stable, repeatable results, we implement the exploit as a proof-of-concept Android application that issues KCI commands through the standard EdgeTPU host interface and execute multiple dump attempts targeting address `0x0FF23750`, separated by a 10-second interval. Between the multiple attempts, a camera application actively uses the TPU for

inference, demonstrating that the oracle operates without disrupting concurrent TPU workloads. Both attempts produce identical DFAR values (`0x43AE1826`), yielding the same recovered cookie `0x34BBE0D6`. This confirms that (1) the primitive is deterministic and reproducible, (2) the leaked value remains stable within a single TPU session as expected from the boot-time derivation, and (3) the attack succeeds while the TPU concurrently serves legitimate inference requests.

This case study uses one deserialization path, but the underlying failure is the unchecked promotion of host-controlled offsets into trusted parser pointers. §5 asks whether that promotion recurs across the reconstructed KCI command-parsing surface.

5 Fault-Oracle Gadgets Detection

To understand how broadly the Fault-Oracle pattern appears across the KCI command-parsing surface, we build a lightweight, Fault-Oracle attack specific static scanner. The goal of the scanner is to systematically triage command handlers for the specific pattern identified in §4: attacker-controlled 32-bit fields that are promoted into derived pointers without sufficient range checks.

The scanner combines taint tracking with coarse interval reasoning. Taint tracking identifies values that originate from the host-controlled command payload and flow into memory-access addresses. Interval reasoning is used to recognize simple local guards and filter cases where the derived value is provably bounded away from protected memory. This design keeps the analysis lightweight while still capturing the key condition needed by the attack.

5.1 Taint Tracking

We first instantiate a standard taint-analysis framework for the Fault-Oracle pattern. The analysis defines three components: sources, which introduce attacker-controlled data; propagation rules, which track how that data flows through the handler; and sinks, where tainted values influence memory-access addresses.

Implementation. We implement the scanner as a forward, intra-procedural, worklist-based dataflow analysis over Hex-Rays microcode, the decompiler's intermediate representation (IR). We analyze microcode rather than raw ARM instructions because the IR normalizes many instruction-level details while preserving enough structure for local dataflow reasoning. We run the analysis at MMAT_CALLS, where call boundaries remain explicit but many ARM Thumb-2 idioms have already been simplified. The analysis is applied independently to each recovered KCI handler.

Taint Sources. For each KCI handler, we identify the micro-register that holds the base address of the incoming command payload at handler entry, using the recovered KCI calling convention. We mark this payload pointer as tainted because it gives the handler access to host-controlled command bytes. From this seed, the scanner tracks values loaded from the command buffer and computations derived from those values.

Taint Sinks. The scanner treats microcode loads and stores as potential sinks. A memory access is reported as a candidate only when its effective address depends on a host-controlled value that can plausibly participate in Fault-Oracle-style pointer derivation. In particular, we look for cases where a command-derived full-width value contributes to an address that is not locally proven to remain in a safe range. Such sites match the attack pattern in §4: a serialized field is interpreted as part of a trusted firmware pointer computation.

Taint Propagation. We first apply conventional taint propagation and then refine it with interval analysis to suppress false positives. Under the standard rules, a value becomes tainted when it is loaded from a tainted address or computed from one or more tainted operands. In our setting, taint therefore flows through copies, loads, and pointer arithmetic rooted in the initial payload register.

The Over-Approximation. However, directly applying conventional taint analysis to the KCI handlers is too coarse for identifying Fault-Oracle gadgets. Ordinary taint marks any value influenced by the command buffer, but not every tainted memory access gives the attacker full 32-bit control over a derived address. To distinguish real gadgets from benign tainted accesses, we therefore need coarse range information about the operands and effective addresses of memory-access instructions. Without such information, conventional taint tracking over-approximates the Fault-Oracle condition and introduces false positives. To address this problem, the next subsection refines taint propagation with interval analysis.

5.2 Taint Propagation with Interval Analysis

To suppress the false positives identified in §5.1, we refine the taint analysis in two ways. First, we distinguish ordinary taint from *derived taint*, which is reserved for full-width attacker-controlled values that can participate in pointer derivation and arithmetic wraparound. Second, we track a coarse unsigned 32-bit interval for each value, so that simple local guards can be recognized during the analysis. Together, these refinements let us focus on attacker-controlled values that can actually drive a Fault-Oracle gadget.

Width-Aware Derived Taint. A value is marked as *derived* only when it is both tainted and wide enough to provide meaningful 32-bit control over a derived address. The initial payload register is therefore *tainted* but not *derived*, because it is a pointer to attacker-controlled payload memory rather than a command-controlled address value. A value becomes *derived* when the handler reads a 4-byte word from a tainted address. By contrast, values read by 1-byte and 2-byte loads remain ordinary taint, since such narrow fields cannot by themselves express a wrap-capable 32-bit address.

Once a value becomes *derived*, we preserve that mark through later propagation unless an operation clearly removes the attacker's full 32-bit control over the value. The main sanitization case is when an operation such as masking proves that the result stays below B_{LO}. We use B_{LO} because it denotes the lower bound of the protected-memory range relevant to Fault-Oracle. Such a value may still be tainted, since it remains influenced by attacker input, but it can no longer steer a pointer dereference into protected memory and is therefore no longer considered derived.

Interval Analysis and Guard Refinement. To recognize when a tainted value is already bounded away from protected memory, we additionally pair each tracked value with a coarse unsigned 32-bit interval. Constants yield singleton intervals, copies preserve intervals, and simple addition and subtraction are interpreted conservatively in this abstract domain. When an operation is not modeled precisely, we over-approximate by assigning the top interval. This preserves soundness at the cost of precision.

The interval state is further refined by control flow. When a branch compares a tracked value against a constant, such as $x < k$ or $x \geq k$, we narrow the interval on each outgoing edge according to the branch outcome. For example, along an edge where $x < B_{LO}$ holds, we refine the interval of x to remain below the protected-memory threshold. When the condition is more complicated or cannot be represented precisely in the interval domain, we leave the interval unchanged on that edge.

Sink Filtering and Reporting. With these refinements, a memory access becomes a candidate Fault-Oracle gadget only when its effective address depends on a derived value and the local interval information cannot prove that value bounded below B_{LO}. This keeps the analysis focused on accesses whose addresses may still be steered by attacker-controlled data into protected memory. It filters out ordinary command-buffer reads, narrow tainted fields, and accesses whose addresses are already proven bounded below B_{LO}. We handle guards conservatively. The analysis removes a candidate only when a check clearly proves that the derived address cannot reach protected memory. Otherwise, the site remains a candidate. To avoid double-counting, we report candidates by the 32-bit load that first introduced the derived value. If one attacker-controlled field influences several nearby loads or stores, we count them once and keep the earliest downstream dereference as the witness.

6 Evaluation

We evaluate the Fault-Oracle primitive and the static scanner through two quantitative research questions, then close with a qualitative downstream analysis case study.

- **RQ1 (Memory-Recovery Effectiveness):** Can Fault-Oracle primitive recover protected TPU memory windows completely, validate the recovered contents, and do so at practical cost?
- **RQ2 (Static Scanner Effectiveness):** How many Fault-Oracle sites does the static scanner identify, and how many are structurally valid and operationally reachable?

Beyond these quantitative measurements, §6.4 shows that the recovered on-chip executable image is useful beyond raw disclosure: by splicing the dumped runtime region into the static firmware image, we repair relocation-broken call targets and switch tables, enabling standard decompilation of a handler that the pre-relocation image alone cannot recover. §6.5 then demonstrates a complementary consequence of runtime-state disclosure: recovered protected code and live runtime SRAM can facilitate vulnerability root-cause analysis and reliable exploit development.

6.1 Experimental Setup

Our experiments were initially conducted on a Google Pixel 9a (tegu) mobile phone running Android 16 with EdgeTPU firmware version `zpr_tpu_20250512.0.0`. After obtaining a Pixel 10a production device, we extracted its EdgeTPU firmware image and verified that it is byte-identical to the firmware used on the Pixel 9a. We further confirmed that the relevant KCI interface and firmware behavior are identical across the two devices. Both evaluated handsets, the Pixel 9a (2025 Q2) and the Pixel 10a (2026 Q1), are built on the Google Tensor G4 SoC. Due to device availability, we did not evaluate Tensor G5–based devices, including the Pixel 10 and Pixel 10 Pro (2025 Q3).

On top of this platform, to emulate a compromised CPU-side kernel, we root the test device and deploy the exploit as a user-space application that submits crafted KCI commands to the EdgeTPU through the standard host interface. Root access further permits the application to read the log ringbuffer of the EdgeTPU directly from kernel memory, which is equivalent to the capability a kernel-level attacker would have natively. For the static analysis, we perform the analysis in IDA Pro 9.2 with IDAPython on the disassembled firmware image extracted from the Google Pixel 9a.

6.2 Results for RQ1

To answer RQ1, we evaluate memory recovery effectiveness via the fault oracle along three dimensions: its *coverage*, *accuracy*, and end-to-end *cost*. The evaluation targets two protected dump windows inside the post-initialization isolation boundary as mentioned in §3.1: the *on-chip executable region* and the *firmware-image subrange within secure DRAM*.

The on-chip executable region spans `0x0FF00000–0x0FF0E4DF` (14,648 aligned 32-bit words, 57.2 KiB) and holds the live TPU-MCU runtime code behind that isolation boundary. This region is the attacker's primary disclosure target: a dump of this region reveals the live TPU-MCU instruction stream that remains hidden behind the post-initialization isolation boundary. With these code bytes, an attacker can analyze the protected image offline, recovering the structure of command handlers, parser state transitions, validation checks, and dispatch logic that are otherwise opaque to the host. This turns a memory-disclosure primitive into a post-exploitation capability: an attacker can use the recovered runtime code to search for deeper firmware-only memory-safety or logic bugs and to craft follow-on inputs that reach those device-side code paths more reliably. As reconstructed in §3.1, the full secure-DRAM mapping spans `0x10000000–0x10120000`; within it, our DRAM evaluation targets the firmware-image subrange `0x10000000–0x1000EA80` (15,009 aligned 32-bit words). During the load-then-lock path, the TPU loader materializes the publicly distributed firmware binary into this subrange and may apply loader/linker fixups before the image is locked and executed; we therefore use the distributed binary as a static byte-level pre-load reference for exact-match checking. We use this subrange primarily as a ground-truth-backed target: even though the pre-load image is publicly available, recovering its locked in-memory copy lets us quantify byte-level agreement against a known reference inside a protected memory window. No analogous static reference exists for the live on-chip executable region.

Together, these two windows exercise complementary validation settings. The secure-DRAM subrange serves as a reference-backed target for measuring byte-level recovery accuracy, while the on-chip executable region serves as the no-reference target that captures the practical disclosure consequence: recovery of protected live runtime code that the host cannot otherwise inspect after initialization.

Memory-Recovery Coverage. We measure coverage as the fraction of aligned 32-bit words in each target dump window for which the fault oracle produces a recovered value. For each aligned address we issue one dump query as described in §4; when the aligned pathway fails to fault cleanly, we activate the sliding-offset fallback (§4.2). The fallback trades per-word pre-load-reference verifiability

Table 2: Exact-match accuracy over the 15,009-word secure-DRAM firmware-image subrange: aligned-only pipeline vs. pipeline augmented with the sliding-offset fallback.

	Baseline	+ Sliding-offset	Δ
Exact match (pre-load ref.)	98.93%	**99.03**%	+0.10 %
Recovered-word coverage	99.73%	**100.00**%	+0.27 %
Residual gaps	41	0	−41
Discrepancies	119	145	+26

Table 3: Structural patterns among the 145 discrepancies between the dumped secure-DRAM firmware-image subrange and the pre-load firmware binary.

Pattern	Count	Interpretation
PC-relative branch immediates	92	Branch rebasing
Long-branch veneers/trampolines	9	Linker/loader stubs
Cross-region pointer-table slots	39	Runtime pointer population
Unattributed	5	Possible runtime-updated variables

for guaranteed gap-free recovery, since a single missing word inside a basic block is sufficient to break decompilation of the enclosing function.

On the secure-DRAM firmware-image subrange, the aligned pipeline recovers 14,968 of 15,009 words (99.73 %); the sliding-offset fallback resolves all 41 residual gaps, bringing coverage to **100** %. On the on-chip region, the aligned pipeline recovers 14,645 of 14,648 words (99.98 %); the fallback recovers the remaining 3 words, likewise reaching **100** %. The fault oracle therefore discloses the entirety of both evaluated protected dump windows.

Accuracy of Memory Recovery. Accuracy is evaluated along two methodologies dictated by whether a static reference exists. For the secure-DRAM firmware-image subrange, we perform a direct byte-level comparison against the authenticated firmware binary as a pre-load reference. For the on-chip region, no static image exists by construction, so we substitute a structural-fidelity proxy: a faithful dump of an executable segment must remain fully analyzable by an off-the-shelf analysis pipeline, whereas injected errors propagate deterministically into undecodable instructions, dangling branches, or orphan bytes.

Protected Firmware Region (Secure DRAM). The secure-DRAM firmware-image subrange from `0x10000000` to `0x1000EA80` contains 15,009 aligned 32-bit words from the authenticated firmware image. Because we possess the original firmware binary, this subrange provides a static byte-level reference for direct comparison. However, the load-then-lock path may apply loader/linker fixups before the image is locked or executed. We therefore treat the static binary as a conservative pre-load reference, not as an absolute runtime image.

Table 2 reports the result of comparing each recovered word against the pre-load reference, both for the aligned-only baseline and for the pipeline augmented with the sliding-offset fallback. With the fallback enabled, exact agreement reaches 14,864/15,009 words, or **99.03** %, while recovered-word coverage reaches **100** %.

Table 3 summarizes the structural attribution of the 145 discrepancies between the recovered secure-DRAM subrange and the pre-load reference. The discrepancies are highly structured: 92 are PC-relative branch immediates, 9 are long-branch veneer/trampoline sites, 39 are cross-region pointer-table slots, and 5 remain unattributed. If the discrepancies came from the recovery process itself rather than loader/linker fixups, they would have no reason to selectively affect relocation-sensitive instruction and data sites; we would instead expect scattered illegal opcodes, invalid control-flow targets, or mismatches with no shared pointer/relocation pattern. Instead, 140/145 discrepancies fall into structured relocation-sensitive categories, exactly the kinds of locations expected to differ between a pre-load firmware binary and a loader-modified runtime image. Only five words remain outside these classes. A plausible source is runtime-updated state: these locations may be used to store variables that the firmware initializes or mutates after load, so they can differ from the static pre-load binary without implying a dump error. Appendix C gives the opcode- and pointer-level attribution details for the structured categories. Thus, the secure-DRAM firmware-image experiment gives two complementary signals. The conservative byte-level comparison against the pre-load firmware binary yields 99.03 % exact-match accuracy at 100 % coverage over this subrange. At the same time, 15,004/15,009 words either match the pre-load reference or fall into relocation-sensitive categories, leaving an unattributed discrepancy rate of only 5/15,009 = 0.03 % (99.97 % attributable or matching), which indicates that the recovered image corresponds to the loader-modified runtime image rather than discrepancies caused by the recovery process.

On-Chip Executable Region. We next target the on-chip executable region of the live TPU runtime. The logical dump window spans 58,592 bytes (57.2 KiB), or 14,648 aligned 32-bit words. Unlike the secure-DRAM firmware-image subrange, this region has no static byte-level reference: it is the live TPU runtime relocated into protected on-chip memory, is not present as a host-visible static binary, and recovering this image is the objective of the attack. Direct byte-level comparison is therefore infeasible by construction.

We validate the dump through executable self-consistency. A faithful dump should load as a coherent ARM Thumb-2 executable under IDA Pro or another off-the-shelf analysis pipeline. Byte-level corruption would be expected to surface as malformed instructions, invalid branch or call targets, broken control-flow, or decompilation failures.

The oracle recovers all 14,648 aligned words in the on-chip executable region, achieving **100 %** coverage; the aligned-only baseline recovers 14,645 words (99.98 %), and only three words require the sliding-offset fallback. Loading the recovered bytes into IDA Pro as ARM Thumb-2 yields a structurally coherent executable (Table 4). The strongest signal is control-flow closure: 99.94 % of `BL`/`BLX` call targets and 97.99 % of B-family branch targets resolve to valid destinations. The supporting instruction- and function-level metrics are consistent with the same conclusion: 99.99 % of bytes are classified as code or data, 99.67 % of instructions fully decode, and 96.86 % of non-trivial functions decompile without manual repair.

These results provide structural evidence that the recovered bytes are consistent with a coherent executable image and preserve

Table 4: Structural validation of the dumped on-chip executable image.

Metric	Rate	Denominator
Bytes classified as code or data	99.99 %	58,592 bytes
Instructions fully decoded	99.67 %	20,816 insns
BL/BLX targets resolved	99.94 %	1,602 calls
B-family targets resolved	97.99 %	1,094 branches
Non-trivial functions decompiled	96.86 %	159 functions

Table 5: Fault-Oracle sites per handler. *Structurally valid* means the site satisfies the three-part primitive definition; *Reachable* means the parsing path can be exercised on the tested Pixel 9a. The dispatcher is the case study in §4.2.

Handler	Reported	Structurally valid	Reachable
Setup	2	2	2
Data transfer	2	2	0†
Teardown	1	1	1
Registration	3	3	3
Execution	15	15	15
Total	**23**	**23**	**21**

†The two data-transfer sites are guarded by a runtime check for a DMA copy engine absent on the tested Pixel 9a; they are structurally valid but configuration-gated on this hardware revision.

the instruction-level and control-flow fidelity needed for downstream vulnerability analysis. They do not assert perfect byte-level correctness for a region with no static byte-level reference, but they show no indication of random byte-level corruption.

Cost of Memory Recovery. An aligned 4-byte dump query whose dump-log entry contains a DFAR completes in 13.4 ms on average, after which the word is computed as P_2 − DFAR. If the dump log marks the word as `cache_unstable`, the dumper reads that word three times, costing ≈ 40 ms. When the aligned query produces no DFAR entry in the dump log, the sliding-offset fallback reconstructs the word from overlapping offset reads at a cost of ≈ 27–54 ms. Across both evaluated protected dump windows (118,628 B in total), the Fault-Oracle sustains an end-to-end disclosure throughput of ≈ 233 B/s (≈ 14 KiB/min) over the standard KCI submission interface; Appendix D reports the per-category breakdown together with the on-chip residual-case analysis.

6.3 Results for RQ2

Table 5 summarizes the static scanner results across the five inner command handlers. For each handler, it reports the number of deduplicated Fault-Oracle sites, how many satisfy the structural primitive definition, and how many are operationally reachable on the tested Pixel 9a.

Across the five inner command handlers, the scanner reports **23 deduplicated Fault-Oracle sites**. We evaluate each report along two axes. A site is *structurally valid* if it satisfies the three-part primitive definition: a 32-bit load from the attacker-controlled command buffer, pointer arithmetic incorporating that value, and an unguarded downstream dereference. A site is *operationally reachable* if the corresponding parsing path can be exercised on the tested Pixel 9a. For each site counted as reachable, we crafted a syntactically valid KCI request that drives the handler to the candidate sink and required the exported DFAR to match the expected faulting address induced by that request. Manual audit finds that all 23 reports are structurally valid, giving **23/23 structural precision**; **21/23** are operationally reachable and underlie the end-to-end dump evaluation of §6.2. The remaining two sites are in the data-transfer `VIICopyData` handler. They are guarded by a runtime check for a DMA copy engine absent on Pixel 9a, so they are structurally valid in the shipped binary but configuration-gated on this hardware revision, not scanner false positives.

The Execution handler accounts for 15 of the 23 sites because it parses variable-length tensor lists through multiple nested indirections, re-reading the vtable pointer at nine distinct program points and resolving six nested tables for input/output descriptors. The other handlers are shallower but still contain valid sites. Microcode generation plus fixpoint analysis across the five handlers completes in under two seconds on a 2024 Apple M3 Pro laptop.

The structural precision is due to three checks that suppress the main false-positive modes we encountered in this firmware. First, the two-level taint lattice separates the attacker-controlled payload pointer from 32-bit values loaded through it, so ordinary flatbuffer parsing reads are not reported as gadgets. Second, the 32-bit width gate promotes only 32-bit attacker-controlled loads to wrap-capable values, excluding vtable-length fields and other narrow loads that cannot cross the 32-bit address space. Third, the interval domain recognizes semantic pointer bounds, rather than matching constants syntactically, and therefore distinguishes secure-region checks from small field-length comparisons. Consistent with these checks, manual inspection found no reported site caused by taint over-propagation, a downstream bounds check, or a narrow/truncated load that removes wrap capability.

6.4 Downstream Utility: Relocation-Aware Analysis

The structural metrics in Table 4 show that the on-chip dump forms a coherent executable image. We next demonstrate why this matters for downstream analysis. The recovered image supplies the runtime layout missing from the static firmware binary, allowing standard analysis tools to resolve relocation-broken call targets and switch tables. We illustrate this on the relocated handler `sub_100056CA`, which is referenced from the KCI dispatcher and fails to decompile usefully from the static DRAM image alone. Figure 3 summarizes the before/after effect of adding the dumped on-chip image to the IDA database.

Failure on the unrelocated image. Loaded into IDA from the static DRAM firmware image alone, `sub_100056CA` fails to decompile, and a switch-case analysis failure was reported. Two structural causes underlie the failure. (i) Several `BL` call sites within the function carry PC-relative targets that, once resolved, fall outside the DRAM-mapped firmware range `0x10000000–0x10120000`: the original targets reside in the on-chip executable region near `0x0FF00000` that the TPU MCU populates only at runtime through relocation, so in the static image these targets dangle into unmapped

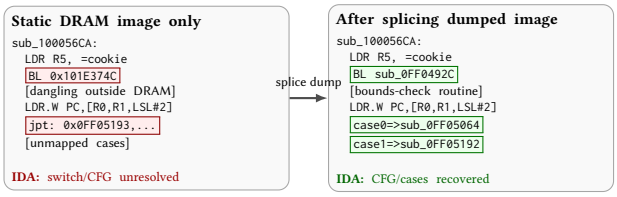


Figure 3: Relocation-aware recovery of `sub_100056CA`. Static DRAM analysis leaves a relocated `BL` and switch-table entries dangling; splicing the dumped on-chip executable at its runtime address resolves the call and case targets, enabling CFG recovery and decompilation.

memory. (ii) The function's switch-case dispatch indexes a jump table whose base is a fixed pointer into the same on-chip region; without the table, IDA cannot enumerate the dispatch successors and abandons control-flow graph construction at the dispatch site. The combined effect is that neither control nor data flow can be tracked past the first relocated `BL` or the switch dispatch, blocking useful decompilation. We confirm that the obstruction is not tool-specific: Ghidra accepts the same function and emits pseudocode, but every `BL` collapses into a `goto` into unmapped addresses and every switch branch into an unresolved indirect jump, so the decompiled output is behaviorally equivalent to the raw disassembly and reveals neither the handler's call structure nor dispatch logic.

Recovery via the dump primitive. We dump the on-chip executable region using the Fault-Oracle primitive of §4 and splice it into the static firmware image at the runtime load address recovered in §3.1, reproducing offline the post-relocation memory layout that the TPU MCU establishes at boot. Re-running auto-analysis on the spliced image, IDA resolves every previously dangling target in this handler. The `BL` site that earlier dangled into unmapped DRAM now lands on a bounds-check routine in the on-chip region, immediately revealing what the handler validates before proceeding; the switch-case jump table, now a valid pointer into the spliced on-chip segment, yields an enumerable target set that allows IDA to complete CFG construction. Both disassembly and Hex-Rays decompilation then succeed without manual intervention, producing pseudocode whose call structure and dispatch logic match the function's runtime behavior. This concrete case demonstrates that the primitive is not only a disclosure channel: in at least this relocation-broken handler, the runtime dump supplies the missing analysis context needed to recover standard control flow and decompiled logic. It therefore materially lowers the cost of subsequent vulnerability analysis on the EdgeTPU runtime without requiring manual reconstruction of the relocated on-chip layout.

6.5 Vulnerability Analysis and Case Study

This section examines how Fault-Oracle supports downstream vulnerability analysis and exploitation. As an end-to-end case study, we analyze a previously unknown firmware memory-corruption vulnerability independently discovered by a concurrent firmware fuzzing project, which initially exposed only an opaque crash. During follow-up analysis and exploitation, Fault-Oracle was used to recover the protected relocated executable and live SRAM, enabling us to identify the root cause and reconstruct the runtime object layout required to achieve reliable code execution on the TPU MCU.

Recovering the root cause. The crash artifact identified a reproducible failure in `ExecuteGraphCommand` and recorded the triggering mutation, host-side buffer mappings, faulting PC, and stack trace. However, because the faulting function had been relocated into protected on-chip SRAM, the static firmware image could not resolve the corresponding runtime code, and the host could not inspect the allocator state that determined the destination address.

Using Fault-Oracle, we recovered the relocated executable containing the faulting PC and the corresponding live SRAM state. Mapping the crash PC into the recovered image revealed that the failure occurred along a memory-copy path. Tracing the destination pointer backward showed that it came from a fallback allocator initialized earlier by `SetupCommand`. The recovered allocator state, together with the host-side mapping recorded during fuzzing, exposed the mismatch: `SetupCommand` validates the scratch-buffer base address, but not whether the full declared range is backed by the IOMMU mapping. This allows the fallback allocator to expose a logical allocation range larger than the host-backed mapping. When an allocation extends past the 32-bit device-address boundary, the resulting destination wraps into low on-chip SRAM, turning the allocator mismatch into an attacker-controlled write to protected memory.

From corruption to code execution. The wrapped write establishes a corruption primitive, but not yet a reliable exploit: it shows that attacker-controlled data can reach protected SRAM, but not which runtime object occupies the reachable range, which field should be targeted, or which surrounding state must remain intact. Fault-Oracle removes this uncertainty by exposing the live SRAM contents around the corruption region. From these dumps, we identified a context-switching object whose virtual-function-table pointer lay within the reachable range and determined its placement relative to the wrapped write. Repeated dumps across executions showed the object at the same address with an essentially identical surrounding layout, indicating that this target layout was stable in our experiments. This allowed us to determine the overwrite offset and the neighboring fields that had to be preserved.

Using the recovered layout, we crafted the wrapped copy to overwrite the object's virtual-function-table pointer while preserving the state required for the object to remain usable. We redirected the pointer to a fake virtual-function table in attacker-controlled TPU-accessible memory. When a later path invoked a virtual function on the corrupted object, a function pointer from the forged table will be loaded and transferred control to an attacker-chosen target, which could help attacker obtain code execution on the TPU MCU.

7 Mitigation

Fault-Oracle relies on three conditions: host-controlled values can influence trusted firmware dereferences without sufficient validation, detailed fault metadata makes the resulting accesses observable, and recoverable faults allow the attacker to repeat the process. Mitigations can break each condition, but preventing unsafe pointer derivation is the fundamental fix.

The primary defense is to re-establish the memory boundary whenever firmware derives an address from host-controlled data.

Validation should therefore apply not only to the top-level KCI command buffer, but also to every derived address before it is dereferenced. A common checked resolver can verify that the derived address and the full access range remain within an authorized region using overflow-safe arithmetic, covering relative offsets, field offsets, strides, and other pointer computations. A stronger design is to decode the host-supplied command into a validated internal representation before use, representing nested references as bounded offsets or handles rather than directly promoting attacker-controlled values into firmware pointers.

Protecting diagnostics provides a second line of defense. Fault-Oracle depends on address-bearing exception metadata, particularly DFAR, stack traces, and related runtime information, to convert a failed dereference into an observable value. Production firmware could therefore expose only coarse failure information needed for reliability, while detailed fault addresses and debugging state should be restricted to trusted engineering environments.

Fault handling can further limit repeated queries. Invalid accesses involving protected memory should terminate the offending command or execution context rather than allowing the same context to continue processing requests. Per-context fault budgets, rate limiting, or a TPU reset when isolation cannot be safely restored can further constrain repeated probing. Address space randomization across resets may also make repeated exploitation less reliable.

Scope of the binary analysis. Our scanner is intended primarily as an initial triage tool for locating instances of the specific Fault-Oracle pattern, rather than as a proof that all unsafe deserialization paths have been identified. A complete coverage argument would require precise inter-procedural reasoning over all command formats, indirect calls, firmware configuration paths, and hardware-dependent memory-access behavior. Such completeness is difficult to establish from binary analysis alone, especially when part of the relevant semantic information is available only in source code, build-time configuration, or internal hardware specifications.

8 Related Work

8.1 Accelerator Security

Accelerator security has attracted increasing attention with the recent advances in AI technology and the widespread deployment of ML accelerators in consumer devices. Prior work on GPU attacks has primarily exposed microarchitectural, physical, or fault-injection channels, including cache contention [20, 33], TLB state [14, 32], electromagnetic emanations [31], rendered-content residue [20], and DRAM-level fault injection [8, 11, 17, 18, 27]. These attacks demonstrate that accelerators can leak information through hardware and physical effects.

Another line of work studies memory safety and firmware security in GPU/NPU software stacks. Guo et al. [9] show that classic exploitation primitives, including buffer overflows, code injection, and ROP, also arise on GPUs, while GPU-Fuzz [16] finds boundary-condition memory errors in deep-learning GPU kernels. CrossFire [34] fuzzes Apple Silicon's driver-mediated cross-XPU interface and finds vulnerabilities that reach GPU/NPU firmware and kernel drivers. MOLE [19] shows that replacing ARM Mali GPU MCU firmware can break a GPU TEE. These works expose important accelerator software risks, but not in authenticated TPU MCU firmware. Our work instead shows how trusted Pixel EdgeTPU firmware can turn remaining host-visible interfaces into a persistent protected-memory disclosure primitive.

To the best of our knowledge, TPUXtract [15] is the only prior work specifically targeting TPU security. It studies the Google Coral Edge TPU as a black-box target and extracts model hyperparameters from electromagnetic traces. Our work instead directly analyzes the Pixel EdgeTPU's isolation boundary and protection.

8.2 Trusted Accelerator Isolation

Several systems extend trusted execution beyond CPUs to protect accelerator workloads. Graviton, HIX, and Telekine protect GPU code, data, and CPU–GPU communication from an untrusted host stack [12, 13, 26]. StrongBox, CAGE, and ACAI extend similar protection to mobile and integrated accelerators such as GPUs and FPGAs [6, 24, 28]. These systems aim to keep protected accelerator workloads hidden from an untrusted OS, driver, or hypervisor.

Beyond GPUs, several systems study TEEs for AI-specific accelerators. IPU Trusted Extensions move confidential-computing support into an AI accelerator itself, with the goal of reducing the host's visibility into accelerator workloads and model data [25]. sNPU studies TEE support for integrated NPUs, protecting NPU workloads and memory while trusted and untrusted software share the accelerator [7]. These designs highlight an important trend: as ML accelerators become security-sensitive execution targets, the TEE boundary must extend beyond CPU memory and include device-local runtime state, accelerator memory, command submission paths, and device-side control logic.

The closest attack-side work is MOLE, which shows that a GPU-embedded MCU can compromise GPU TEEs when attacker-controlled firmware is accepted [19]. Our threat model and failure mode are different: we do not replace, patch, or rehost the TPU firmware. Instead, we attack the authenticated firmware loaded through the platform's normal protection path.

In a nutshell, our work is complementary to these accelerator-TEE designs. Rather than proposing a new trusted accelerator architecture, we analyze a deployed mobile TPU that is already treated as trusted hardware. We show that the hardware boundary can remain intact while trusted firmware still breaks confidentiality through residual command and diagnostic channels. The gap lies in internal boundary enforcement, where firmware validates the top-level host command buffer but later derives pointers from attacker-controlled offsets without rechecking the memory boundary.

9 Conclusion

In this work, we analyze Pixel EdgeTPU security and show that protected TPU runtime memory can be disclosed without bypassing GSC-enforced access controls. We uncover trusted deserialization faults and develop Fault-Oracle, a persistent arbitrary-read attack that composes crafted KCI inputs with fault-log metadata. Using a lightweight taint-based scanner, we identify 23 gadgets and recover runtime memory from a production Pixel device. More broadly, the work highlights the need to analyze and protect command parsing and diagnostic interfaces as part of the security boundary. We responsibly disclosed the identified vulnerabilities to Google.

References

[1] Android Open Source Project. 2026. Security updates and resources. https://source.android.com/docs/security/overview/updates-resources. Android Open Source Project documentation, accessed April 18, 2026.

[2] Android Open Source Project. 2026. Trusty TEE. https://source.android.com/docs/security/features/trusty. Android Open Source Project documentation. Accessed 2026-03-24.

[3] Asahi Lina. 2021. Reverse Engineering the M1 GPU. https://asahilina.net/gputalk/. Slides, accessed April 18, 2026.

[4] Asahi Linux Project. 2021. Progress Report: August 2021. https://asahilinux.org/2021/08/progress-report-august-2021/. Section "Reverse Engineering DCP", accessed April 18, 2026.

[5] Coral Team. 2020. Products. https://www.coral.ai/products/. Coral website, accessed April 18, 2026.

[6] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. 2022. Strongbox: A gpu tee on arm endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 769–783.

[7] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, and Haibo Chen. 2024. sNPU: Trusted execution environments on integrated NPUs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 708–723.

[8] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *2018 IEEE Symposium on Security and Privacy (SP)*. 195–210. doi:10.1109/SP.2018.00022

[9] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. {GPU} Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. 4033–4050.

[10] Suyog Gupta and Marie White. 2021. Improved On-Device ML on Pixel 6, with Neural Architecture Search. https://research.google/blog/improved-on-device-ml-on-pixel-6-with-neural-architecture-search/. Google Research blog, accessed April 18, 2026.

[11] Yichang Hu, Noah Brown, Yuhang Chen, Joshua Bakita, Tianlong Chen, Daniel Genkin, and Andrew Kwong. 2026. GDDRHammer: Greatly Disturbing DRAM Rows — Cross-Component Rowhammer Attacks from Modern GPUs. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA.

[12] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. 2020. Telekine: Secure computing with cloud {GPUs}. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 817–833.

[13] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 455–468.

[14] Hongyue Jin, Yanan Guo, and Zhenkai Zhang. 2026. Exploiting TLBs in Virtualized GPUs for Cross-VM Side-Channel Attacks. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*. San Diego, CA, USA. doi:10.14722/ndss.2026.231480

[15] Ashley Kurian, Anuj Dubey, Ferhat Yaman, and Aydin Aysu. 2025. TPUXtract: An exhaustive hyperparameter extraction framework. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 1 (2025), 78–103.

[16] Zihao Li, Hongyi Lu, Yanan Guo, Zhenkai Zhang, Shuai Wang, and Fengwei Zhang. 2026. GPU-Fuzz: Finding Memory Errors in Deep Learning Frameworks. *arXiv preprint arXiv:2602.10478* (2026).

[17] Chris S Lin, Joyce Qu, and Gururaj Saileshwar. 2025. {GPUHammer}: Rowhammer attacks on {GPU} memories are practical. In *34th USENIX Security Symposium (USENIX Security 25)*. 5719–5738.

[18] Chris S. Lin, Yuqin Yan, Guozhen Ding, Joyce Qu, Joseph Zhu, David Lie, and Gururaj Saileshwar. 2026. GPUBreach: Privilege Escalation Attacks on GPUs using Rowhammer. In *Proceedings of the 47th IEEE Symposium on Security and Privacy (SP '26)*.

[19] Hongyi Lu, Yunjie Deng, Sukarno Mertoguno, Shuai Wang, and Fengwei Zhang. 2025. mole: Breaking gpu tee with gpu-embedded mcu. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 693–707.

[20] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2139–2153.

[21] Rick Osterloh. 2021. Google Tensor debuts on the new Pixel 6 this fall. https://blog.google/products-and-platforms/devices/pixel/google-tensor-debuts-new-pixel-6-fall/. Google The Keyword blog. Accessed 2026-03-24.

[22] Jesse Seed and Brandon Barbello. 2021. Pixel 6: Setting a new standard for mobile security. https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html. Google Security Blog. Accessed 2026-04-15.

[23] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, et al. 2025. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *Comput. Surveys* 57, 11 (2025), 1–39.

[24] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. 2024. {ACAI}: Protecting accelerator execution with arm confidential computing architecture. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3423–3440.

[25] Kapil Vaswani, Stavros Volos, Cédric Fournet, Antonio Nino Diaz, Ken Gordon, Balaji Vembu, Sam Webster, David Chisnall, Saurabh Kulkarni, Graham Cunningham, et al. 2023. Confidential computing within an {AI} accelerator. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 501–518.

[26] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted execution environments on {GPUs}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 681–696.

[27] Junpeng Wan, Yanan Guo, Zhi Zhang, Zhuo Li, Dave (Jing) Tian, and Zhenkai Zhang. 2026. GeForge: Hammering GDDR Memory to Forge GPU Page Tables for Fun and Profit. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA.

[28] Chenxu Wang, Fengwei Zhang, Yunjie Deng, Kevin Leach, Jiannong Cao, Zhenyu Ning, Shoumeng Yan, and Zhengyu He. 2024. CAGE: Complementing Arm CCA with GPU Extensions.. In *NDSS*.

[29] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. 2025. Empowering edge intelligence: A comprehensive survey on on-device ai models. *Comput. Surveys* 57, 9 (2025), 1–39.

[30] Amir Yazdanbakhsh, Berkin Akin, and Kiran K. Seshadri. 2021. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. arXiv:2102.10423. Google Research publication page: https://research.google/pubs/an-evaluation-of-edge-tpu-accelerators-for-convolutional-neural-networks/. Accessed 2026-03-24.

[31] Zihao Zhan, Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xenofon Koutsoukos. 2022. Graphics Peeping Unit: Exploiting EM Side-Channel Information of GPUs to Eavesdrop on Your Neighbors. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1440–1457. doi:10.1109/SP46214.2022.9833773

[32] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 960–974.

[33] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. 2024. {Invalidate+ Compare}: A {Timer-Free}{GPU} Cache Attack Primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2101–2118.

[34] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. 2024. Crossfire: Fuzzing macos cross-xpu memory on apple silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3749–3762.

A Open Science

The artifacts needed to reproduce our results comprise the proof-of-concept, static analyzer, per-site KCI inputs, recovered memory images, and evaluation scripts. We defer their public release pending coordinated remediation and disclosure with Google to avoid exposing unpatched devices. We informed the program chairs of this constraint before submission. Following remediation and coordination with Google, we plan to publish the artifacts in a persistent public archive and add its reference to an updated public version of this paper.

B Ethical Considerations

We reported the fault-oracle primitive and all 23 scanner-identified sites to Google through the Android VRP; Google acknowledged the report and confirmed reward eligibility. We coordinate disclosure with Google and Google has also since developed and deployed remediation addressing the reported issues. All experiments used researcher-owned devices, accessed no user data, and involved no third-party systems. Root access served only to emulate the kernel-level attacker specified in our threat model.

C Detailed Secure-DRAM Discrepancy Attribution

This appendix gives the details behind the structured discrepancy analysis in Table 3. For completeness, the aligned-only pathway left 41 addresses without a DFAR entry in the dump log; these addresses spanned 38 functions. The sliding-offset fallback closed all 41 gaps, with 15/41 = 36.59 % also matching the pre-load reference exactly.

PC-relative branch immediates. The largest class consists of 92 Thumb-32 `BL/B.W` branch-immediate words. These instructions encode their target as a split PC-relative immediate across two halfwords. Changing the loaded address of either the caller or target can therefore change the instruction word while preserving the same semantic branch target. All 92 discrepancies lie at addresses whose two halfwords match the Thumb-32 `BL/B.W` prefix pattern: the high halfword matches the long-branch class and the low halfword matches the corresponding branch continuation halfword. This concentration is consistent with branch rebasing rather than scattered byte-level modification.

Long-branch veneers and trampolines. Nine discrepancies fall in two compact ranges. At these offsets, the pre-load firmware binary stores strictly increasing 32-bit pointer-like values. The dumped image instead contains sequential ARM-mode `MOV` encodings. The sequence matches the scratch-register shuffling used by long-branch trampoline code, the kind of stub a linker or loader emits when a direct `BL` cannot reach its final target after layout.

Cross-region pointer-table slots. The 39 data-side discrepancies occur in pointer-table slots that bridge the secure-DRAM image and the independently dumped on-chip executable region. In the pre-load binary, 15 of these slots are zero and 24 hold small runtime-handle-like values, typically in the `0x000eXXXX` range. In the dump, the corresponding words are resolved Thumb pointers into the `0x0FF00000` region in 80 % of the slots: their low bit is set, contiguous entries advance by +4, and each target lands on a legitimate Thumb function entry. The targets are corroborated by the independently dumped on-chip image: the `0x0FF00000` addresses recovered from these slots correspond to function entries identified by IDA's auto-analysis in that image. This agreement links the two independently recovered regions and supports runtime pointer population as the explanation for this cluster.

D On-Chip Executable Validation Details

Memory layout. The on-chip executable image is the live TPU runtime relocated into protected on-chip memory. The MPU exposes it through two read-execute regions. Region 15 spans `0x0FF00000`–`0x0FF0405F` (16.094 KiB) and is privileged-only. Region 3 spans `0x0FF04080`–`0x0FF0E4DF` (41.094 KiB) and is accessible in both privileged and unprivileged modes. The two regions are contiguous modulo a 32-byte reserved gap, so the evaluation treats them as one logical dump window of 58,592 bytes (57.2 KiB), or 14,648 aligned 32-bit words.

Structural metrics. We load the recovered bytes into IDA and use the default auto-analysis output, without manually forcing additional function starts or repairing control flow. Table 4 reports the primary metrics. Canonical prologues occur in 96.60 % of 235 functions and 96.58 % of the 146 `BL/BLX`-confirmed functions. IDA classifies 99.99 % of bytes as either instructions (97.83 %) or literal-pool data (2.16 %); only 7 bytes remain unresolved. The canonical-prologue rates are included as a secondary heuristic, while the main text emphasizes control-flow closure because branch and call targets directly test whether the recovered image forms a coherent executable.

Table 6: Per-word extraction cost by read category, and resulting wall-clock for Region 1 (`0x0FF00000`–`0x0FF0E4DF`) and Region 2 (`0x10000000`–`0x1000EA80`).

Category	Cost	Region 1	Region 2	Combined
Normal (1 read)	13.4 ms	12,475	12,859	25,334
Cache-unstable (3 reads)	40 ms	2,170	2,109	4,279
Sliding-offset fallback	27–54 ms	3	41	44
Total words		14,648	15,009	29,657
Wall-clock		4.2 min	4.3 min	8.5 min
Throughput		232 B/s	233 B/s	232 B/s

Residual cases. The 7 unresolved bytes cluster at literal-pool tails or padding-like locations where IDA cannot cleanly choose between word-aligned data and trailing padding. The non-canonical function entries are mostly expected embedded-code patterns, including `MRC` system-register reads and `NOP` scheduling slots, rather than malformed instruction streams. The near-identical canonical-prologue rate in the BL/BLX-confirmed subset indicates that these entries are not an artifact of unreferenced function discovery. The remaining unresolved branch and call cases are dominated by code–data dual-use words, such as jump-table terminators and in-range literal pools that the toolchain legitimately interleaves with code. We also find no ASCII string of length $\geq$ 20, which is auxiliary evidence that the segment is pure executable code rather than a mixed `.rodata` region. Together, these residual cases are consistent with ordinary ambiguity and do not show the scattered malformed instructions, invalid targets, or broken CFG edges expected from random byte-level corruption.

Extraction cost. An aligned 4-byte dump query whose dump-log entry contains a DFAR completes in 13.4 ms on average, measured via `clock_gettime` around the Fault-Oracle. If the dump log marks the word as `cache_unstable`, the dumper issues three aligned queries at $3 \times 13.4 \approx 40$ ms per word, accounting for 29.2 % of words (4,279 of 29,657) across both target regions. Words whose aligned query produces no DFAR entry in the dump log are rescued by the sliding-offset fallback, which reconstructs the word from overlapping offset reads at $\approx$ 27–54 ms per word. Table 6 gives the per-category breakdown.