I. Discussion

The 2025 WhatsApp–ImageIO chain and its 2021 FORCEDENTRY predecessor demonstrate a consistent architectural pattern: a delivery primitive (messaging protocol bypass) composes with an exploitation primitive (image parser memory corruption) to achieve zero-click code execution. This section examines why the current mitigation stack did not prevent the 2025 chain, what lies beyond the initial RCE, what structural alternatives exist, and the limitations of this study.

A. Why the Mitigation Stack Did Not Prevent This Chain

Section **??** introduced three mitigations deployed on iOS: ASLR, stack canaries, and application sandboxing. None of them target the root cause of CVE-2025-43300.

ASLR randomizes memory layout but does not prevent the out-of-bounds write itself. The write occurs regardless of where the buffer is placed; ASLR only makes it harder to know where to redirect control flow afterward. In practice, a second, milder bug in the same process—an information leak—suffices to defeat ASLR by revealing a code or data address, after which the attacker can construct a valid target for the hijacked pointer. Information leaks in image parsers are common: the same complexity that produces out-of-bounds writes routinely produces out-of-bounds reads [**?**].

Stack canaries are entirely irrelevant: the overflow occurs on the heap, not the stack. Canaries protect saved return addresses; they do not monitor heap metadata or function pointers stored in heap-allocated objects.

Application sandboxing limits what the attacker can do after RCE but does not prevent the RCE itself. The sandbox around WhatsApp restricts file-system and IPC access, but the attacker's initial foothold—code execution inside the process—is already sufficient to exfiltrate data accessible to that process (contacts, messages, media, location) and to attempt a sandbox escape via a separate vulnerability.

Pointer Authentication Codes (PAC), deployed on ARMv8.3+ Apple silicon [**?**], sign pointers with a process-specific secret so that corrupted pointers fail verification before use. PAC raises the cost of control-flow hijacking significantly but has been bypassed in academic and in-the-wild settings, typically by reusing legitimately signed pointers or by targeting code paths that do not enforce PAC on every indirect call.

The pattern is consistent: each mitigation addresses a *symptom* (where code lives, whether the return address is intact, what permissions the process has, whether a pointer is signed) rather than the *root cause* (the program wrote past the end of a buffer because no bounds check existed). So long as the parser is written in a language that permits unchecked memory access, the vulnerability class persists and mitigations can only raise the cost of exploitation, not eliminate it.

B. Beyond RCE: Sandbox Escape and Persistence

A zero-click RCE inside a sandboxed process is not, by itself, full device compromise. The reported FORCEDENTRY and 2025 chains both achieved persistent, cross-application surveillance (call recording, message exfiltration, camera/microphone access), implying that additional stages followed the initial RCE: at minimum, a sandbox escape and a privilege-escalation step. These stages are outside the scope of this paper—they involve kernel vulnerabilities and system-daemon bugs unrelated to the two CVEs studied—but their existence must be acknowledged: the image-parser RCE is the entry point, not the endpoint, of the full attack.

C. Structural Alternatives

If mitigations cannot close the vulnerability class, what can? Two categories of structural change address the root cause.

Memory-safe languages—Rust, Swift, Go, Java—enforce bounds checking, type safety, and lifetime management at compile time or runtime, making out-of-bounds writes impossible in safe code by construction [**?**], [**?**]. Rewriting media parsers in a memory-safe language would eliminate CWE-787 from the attack surface entirely. The industry is already moving in this direction. Apple has begun shipping Swift-based replacements for some system components. Google's Android team has documented that the share of memory-safety vulnerabilities in new Android code fell to near zero in modules rewritten in Rust [**?**]. Most significantly, the Linux kernel—the canonical example of a safety-critical C codebase—merged initial Rust infrastructure in Linux 6.1 (December 2022) and shipped its first production Rust-written driver (a network PHY driver for the Asix ax88796b chipset) in Linux 6.8 (March 2024); Rust was declared a permanent, non-experimental part of the kernel at the 2025 Kernel Maintainer Summit in Tokyo [**?**]. At the distribution level, Ubuntu 26.04 LTS (April 2026) became the first major long-term-support release to ship *sudo-rs*—a memory-safe Rust reimplementation of `sudo`—as the *default* privilege-escalation binary [**?**], replacing a C codebase that has been a high-value target for local privilege escalation for decades. The challenge for media frameworks is scale: ImageIO is a large, performance-critical C/Objective-C codebase with decades of accumulated optimisation, and incremental rewriting is slow—but sudo-rs and the Linux Rust drivers demonstrate that production-grade, security-critical C code can be incrementally replaced.

Hardware-enforced memory safety, specifically ARM Memory Tagging Extension (MTE) on ARMv8.5+, assigns a 4-bit tag to every 16-byte memory granule and checks the tag on every access. A heap overflow that crosses a granule boundary with the wrong tag traps immediately, preventing the corruption from reaching adjacent objects. MTE is deployed on some Android devices but is not yet enabled on iOS. If deployed, MTE would

make the CVE-2025-43300 overflow detectable at the hardware level regardless of the implementation language.

D. Limitations

Three limitations must be acknowledged. First, our analysis of CVE-2025-43300 is second-hand: we rely on Quarkslab's reconstruction from patch diffing [**?**], not on independent reverse engineering. Certain exploitation details (heap-spraying strategy, PAC bypass technique) are inferred, not observed. Second, the Phoenix `heap-two` exercise deliberately omits real-world complications (ASLR, PAC, heap randomization, concurrent allocation); it illustrates the core mechanic, not the full exploitation engineering. Third, we compare only two chains (FORCEDENTRY and the 2025 chain); broader generalization of the "same pattern, different parser" thesis requires analysis of additional cases as they are disclosed.