SECV: Securing Connected Vehicles with Hardware Trust Anchors

Martin Kayondo*‡, Junseung You*‡, Eunmin Kim*, Jiwon Seo†§, Yunheung Paek*§
*ECE and ISRC, Seoul National University
{kymartin, jsyou, emkim}@sor.snu.ac.kr, ypaek@snu.ac.kr
†Department of Cybersecurity, Dankook University
jwseo@dankook.ac.kr

***Abstract*—Modern vehicles integrate Extra-Vehicle Networks (EVNs) with In-Vehicle Networks (IVNs) to support navigation, diagnostics, and over-the-air updates. This convergence introduces an EVN platform as a new source of control messages at the IVN gateway, breaking the traditional assumption that the gateway only filters traffic from simple, isolated, and implicitly trusted legacy ECUs. Instead, the EVN platform hosts a complex EVN manager with a full operating system and multiple applications, greatly enlarging the attack surface: a compromised OS or application can spoof control messages that evade gateway filtering. We present SECV, a runtime security mechanism that enables the IVN gateway to accurately verify EVN-originated control messages even when the EVN manager is compromised. SECV mediates all EVN-to-IVN traffic inside a Trusted Execution Environment (TEE), performs per-application validation, and attaches cryptographic proofs. These proofs are verified by the IVN gateway using a Hardware Security Module (HSM), providing reliable message authentication with low overhead. SECV addresses practical challenges in TEE–HSM trust establishment, real-time mediation, and robust attribution under compromise. Implemented on an automotive SoC with ARM TrustZone and an EVITA-compliant HSM, SECV enforces strong security guarantees with only 6.5% transmission geometric mean overhead and 1.5% additional message loss during extreme communication bursts, effectively mitigating EVN-originated attacks while satisfying real-time constraints.**

I. INTRODUCTION

Modern connected vehicles support advanced features such as navigation assistance, real-time traffic updates, and remote diagnostics, which enhance both safety and passenger experience and rely on continuous data exchange with cloud services, backend servers, user devices, and other vehicles. This need for external connectivity marks an evolutionary shift from legacy automotive systems, built around a closed in-vehicle network (IVN), to an architecture that integrates the IVN with an extra-vehicle network (EVN) to support high-bandwidth data exchange with diverse external entities. This architectural paradigm, now widely adopted by major automotive vendors [41], [71], [79], [81], logically separates internal (IVN) and external (EVN) communications into distinct domains, typically interfaced through dedicated gateway components to meet the growing data demands of modern features. The IVN platform, typically based on microcontrollers and designed for real-time operation, hosts a software stack known as the *IVN gateway*, which oversees IVN traffic and is present even in legacy systems. In contrast, the EVN platform, typically built on application processors optimized for compute-intensive tasks and introduced as part of the connected vehicle paradigm, runs the *EVN manager*, a software stack responsible for handling EVN traffic. The EVN manager and IVN gateway coordinate to mediate interactions between the EVN and IVN domains.

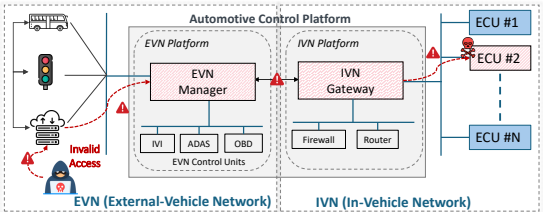


Fig. 1: Remote attackers can leverage EVN reach to hack their way to safety-critical IVN ECUs

Unfortunately, the EVN domain significantly expands the vehicle's attack surface by introducing remote threats. In traditional vehicles, the closed and isolated nature of IVNs largely protected electronic control units (ECUs) from remote compromise, limiting realistic attacks mostly to physical ones [54]. However, as shown in Figure 1, prior work [24], [25], [33], [35], [48], [53], [64], [65], [69], [72], [90] demonstrates that modern platforms allow remote attackers to compromise the EVN manager and launch attacks on the IVN. Because the IVN hosts ECUs responsible for safety-critical functions (e.g., braking, steering, engine control), arbitrary control-message injection enables effective commandeering of the vehicle and poses severe risks to passenger and public safety. These attacks have drawn increasing attention from society, security researchers, and industry practitioners [22], [23], [28], [30], [36], [46], [49], [67], [80], [82], [85]–[88], [91]–[93], [98], [99], [102]–[104], [108], underscoring the need for systematic defenses that constrain EVN-originated threats before they

‡The first two authors contributed equally to this work.
§Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240106
www.ndss-symposium.org

reach the IVN.

Traditionally, automotive security has been enforced by the IVN gateway, which applies static routing and filtering policies to control all in-vehicle communications among simple, trusted ECUs. However, the introduction of the EVN and its complex EVN manager—a powerful, connected node running a full OS with multiple applications—breaks these trust assumptions. Real-world attacks [43], [58], [64], [95], [96], [107] show that a compromised EVN manager can exploit the IVN gateway's lack of knowledge about runtime EVN context (e.g., sender application identity or intent) to inject spoofed control messages that appear legitimate, undermining the gateway's filtering.

To mitigate these attacks, we propose a security mechanism that guarantees the trustworthiness of control messages from the EVN manager, enabling the IVN gateway to process them for verification. Our mechanism mediates all outbound control messages from the EVN manager to ensure that only authorized messages are transmitted. This mediation occurs within a protected execution environment to prevent circumvention, even under OS compromise. Second, the mechanism enforces message transmission policies that determine whether a given application is authorized to send specific control messages. These policies are securely stored in a tamper-resistant environment, with both the policies and their enforcement protected from manipulation—even under OS compromise. Finally, our mechanism enables the IVN gateway to verify these messages efficiently and securely, such that authentication can be performed within the real-time and safety constraints of the automotive system.

We have implemented our security mechanism, named SECV, with hardware-based security primitives [26], [51], [70] widely available on modern automotive platforms. Specifically, SECV builds a trusted *enforcement module* using a Trusted Execution Environment (TEE) [51]—commonly present on EVN platforms. The main tasks of the EVN enforcement module are outbound message authentication, authorization, and spoofing detection. To perform these tasks, it relies on the TEE's secure peripheral access control to intercept all outbound traffic for inspection. Additionally, it leverages the TEE's tamper-resistant runtime environment to securely store and enforce per-application transmission policies based on execution context (e.g., process ID and address space). The module signs authorized messages using a TEE-confined key, securely shared with the IVN gateway for verification. Message signatures provide unforgeable proof to the IVN gateway that the messages are trustworthy and safe to process before being routed to the IVN ECUs. On the IVN side, the IVN gateway verifies each message's signature before routing it to IVN ECUs. The gateway employs a Hardware Security Module (HSM)—mandated on IVN platforms by automotive standards [75], [106]—as its trust anchor. The gateway relies on HSM to attest the EVN enforcement module, securely store the message verification key, and offload cryptographic signature verification to reduce computational overhead and meet real-time and safety requirements. Together, these components on both the EVN and IVN sides collaboratively enforce a layered authentication architecture: the enforcement module ensures that only trusted messages transmitted by the EVN manager reach the IVN gateway, while the IVN gateway independently verifies their authenticity and authorization before applying traditional filtering.

Designing SECV presents three key challenges. First, for the IVN gateway to trust control messages from the EVN manager, it must trust the enforcement module that authorizes them. Since the IVN gateway anchors its trust in an HSM, the HSM must validate the TEE firmware on which the enforcement module runs and establish a secure communication channel with it. However, the HSM lacks direct access to the TEE firmware, and its internally generated keys cannot be exported to establish the shared communication channel. Second, mediating all outbound transmissions through the enforcement module requires it to control communication peripherals—intercepting I/O, routing interrupts securely, and executing peripheral control logic within the Secure World. If implemented naively, this can introduce frequent world switches that incur significant performance overhead, undermining the real-time guarantees required for automotive safety. It also risks bloating the TEE's trusted computing base (TCB) with peripheral control code, increasing the system's attack surface. Third, SECV must reliably associate each outbound control message with its originating application, without executing EVN applications inside the Secure World. This requires extracting and validating runtime context (e.g., process ID and address space) within the TEE, even in the presence of a compromised Normal World OS.

To address the first challenge, SECV introduces a hybrid protocol that combines measured boot with cross-domain attestation and coordinated key provisioning. The EVN platform's root of trust—a lower-layer bootloader—measures the TEE firmware during boot and transmits signed attestation evidence to the HSM via a pre-established asymmetric key relationship. To establish secure communication, the EVN enforcement module receives a random entropy from HSM, and derives a strong communication key which is securely imported into the HSM for synchronized use. To address the second challenge, SECV implements a lightweight TEE–peripheral coordination protocol that minimizes world switches through batching and polling, and further optimizes transmission using zero-copy shared memory channels. To maintain a minimal TCB for peripheral access, SECV avoids porting full drivers into the TEE. Instead, it employs minimal *driverlets*—lightweight, security-critical stubs that execute only trusted memory-mapped I/O (MMIO) operations, delegating all non-sensitive functionality to the Normal World driver. To address the third challenge, SECV establishes *secure message channels* between user-space applications and the EVN enforcement module. Leveraging intra-kernel isolation [34], [56], [60], SECV deploys a minimal trusted kernel component that accurately tracks application identities and securely relays this context to the enforcement module, enabling authenticated message attribution for policy enforcement.

We implement SECV on a modern automotive SoC with

an EVN platform based on ARM Cortex-A processors with TrustZone and an IVN platform based on ARM Cortex-M7 MCUs with an EVITA-compliant HSM [106], where the EVN and IVN communicate over CAN. This architecture reflects current industry practice, as major vendors (e.g., Infineon, NXP, Renesas) deploy ARM-based platforms with integrated TEEs and HSMs, with CAN as the dominant IVN protocol and growing Ethernet use in EVN control units. To evaluate SECV's practicality, we measure runtime overhead, communication performance, and memory footprint, observing a 6.5% EVN-to-IVN transmission geometric mean overhead and 1.5% message loss under high-throughput bursts. These results show that SECV can provide strong security guarantees while meeting modern automotive performance requirements, and a detailed security analysis confirms its effectiveness against remote attackers, compromised OSes, and malicious applications under realistic threat models.

II. Background and Motivation

A. Automotive Architecture and EVN-IVN Communication

Automotive systems feature heterogeneous E/E architectures in which ECUs exchange control messages over the IVN; for example, a speed-control ECU (e.g., cruise control or ADAS) may receive a stop command with a unique message ID from the brake ECU. Traditionally, IVN ECUs run on simple single-threaded MCUs that implement one or a few fixed functions, allowing the IVN gateway to enforce static filtering and routing rules and to use the fixed mapping between message IDs and ECU tasks to deterministically associate each message with its source and promptly block malicious traffic targeting safety-critical systems. However, this model implicitly assumes that all IVN endpoints are simple, fixed-function ECUs. As vehicles adopt the EVN and its EVN manager, the IVN gateway typically remains the primary IVN defense but still treats the EVN manager as just another ECU whose traffic must match traditional filters. In reality, the EVN manager hosts multiple OS-managed applications, each able to transmit through the IVN gateway, while the gateway has no visibility into runtime context or sender provenance. As a result, messages from a compromised EVN manager can satisfy static rules and be forwarded to IVN ECUs, enabling unauthorized control. Moreover, some EVN manager-origin messages are privileged and target the IVN gateway itself, raising the risk of malicious firmware installation and propagation across ECUs.

B. Automotive Attacks and SECV Motivation

1) ***Control Message Paths and Compromise****:* To transmit a control message, an EVN application first prepares the message in a memory buffer and then invokes the network stack via a system call. The stack typically copies the message from the user-space into a kernel-space buffer, where it may further process the message according to the transmission protocol (e.g., header construction) before enqueueing it for transmission. Actual transmission occurs when the message is dequeued and written to the communication peripheral, such as a CAN controller, by the corresponding driver executing in the OS address space. In some cases, particularly with custom peripherals, applications may bypass the standard network stack and interact directly with the communication driver via the `IOCTL` interface, passing the message straight to the driver, which may then perform additional processing or write the message directly to the peripheral. At each stage, the message can be compromised or forged; that is, **(1)** Application layer: a malicious application generates and transmits control messages that impersonate an authorized application. **(2)** Network stack: a compromised stack tampers with messages in transit or injects unauthorized messages into its transmission queues to spoof application provenance. **(3)** Peripheral driver: a compromised driver, which directly controls the communication peripheral, injects attacker-chosen messages directly into peripheral buffers and triggers their transmission.

Requirements for EVN-IVN attack mitigations:

RQ1: All EVN applications must have explicitly defined permissions governing which control messages they are allowed to transmit.

RQ2: Transmission permissions are security-critical and must be protected against unauthorized modification or spoofing.

RQ3: Only authenticated and attested firmware and software are allowed to execute on both EVN and IVN platforms.

RQ4: All security keys must be securely stored and properly managed, and their use must be strictly controlled to prevent unauthorized access or tampering.

RQ5: Messages from the EVN manager must be consistently verified, and the EVN manager must cooperate with the IVN gateway to establish comprehensive trust validation for all EVN–IVN traffic.

2) ***Real-world Attack Examples****:* The attacks described above have been demonstrated by security researchers and industry practitioners on modern production vehicles. Based on an analysis of these published attacks, we derive the core requirements for establishing the EVN manager as the trustworthy source of IVN control messages and, in turn, for defending against EVN-IVN attacks.

Below, we summarize five real-world attack examples (**AE**) and show how unmet requirements (**RQ**) enabled them.

AE1: Tencent Keen Lab vs. Mercedes-Benz (2021). Keen Lab exploited firmware vulnerabilities in the STA8090 Wi-Fi receiver to gain code execution on the head unit (EVN manager) [8]–[10]. Subsequent browser and kernel exploits yielded root privileges on this platform. By impersonating trusted services on the EVN manager, the attackers were able to bypass the IVN gateway and inject CAN messages. They then achieved persistence by modifying the firmware of the telematics control unit (TCU), which is typically attached to the EVN manager. (~~**RQ1**~~, ~~**RQ3**~~).

AE2: Cai et al. vs. BMW (2019). Cai et al. from Tencent

Key observations from attack examples:

AO1: These attacks span over a decade, with multiple cases as recent as 2023–2024, indicating that weaknesses are persistent rather than legacy artifacts.

AO2: In all examined designs, the EVN and IVN remain air-gapped, with the IVN gateway as the single chokepoint. Once compromised, attackers gain complete control.

AO3: Across vendors, attacks follow a common pattern: EVN side entry, EVN manager compromise, and IVN gateway filtering bypass that enables arbitrary IVN control message injection.

AO4: In some Tesla cases, compromising only the EVN manager was sufficient to bypass gateway protections and reach safety-critical ECUs, even without directly modifying the gateway firmware.

exploited vulnerabilities in the TCU's cellular interface to compromise BMW's EVN components, including the In-Vehicle-Infotainment (IVI) system and the EVN manager [57]. A WebKit buffer overflow enabled initial code execution, and a time-of-check-to-time-of-use (TOCTOU) flaw was then used to escalate privileges to root on the EVN manager. The attackers subsequently crafted unchecked control messages that exploited weaknesses in the IVN gateway's filtering, allowing them to issue commands to safety-critical ECUs. (~~**RQ1**~~, ~~**RQ5**~~)

AE3: Miller & Valasek vs. Jeep Cherokee (2015). By exploiting a firmware vulnerability in the TCU [2], the attackers first gainted a foothold on the EVN manager and then pivoted the IVN gateway over the SPI [64] link. They uploaded a malicious firmware image to the gateway, disabling SPI–CAN filtering, and demonstrated remote control of brakes, engine, and transmission at highway speeds. (~~**RQ1**~~, ~~**RQ3**~~, ~~**RQ5**~~)

AE4: Keen Lab vs. Tesla Model S (2016–2017). A browser exploit [1] and kernel escalation [4] gave attackers root on the EVN manager allowing them to reconfigure IVN gateway filters and install malicious firmware, thereby enabling direct CAN access from the EVN side [68]. After Tesla added firmware signing in 2017, the researchers bypassed it by exploiting the update logic to inject and execute unsigned EVN applications. (~~**RQ3**~~)

AE5: Synacktiv vs. Tesla Model 3 (2022–2024). Synacktiv demonstrated a series of full remote compromises via the IVI system [29], [94]–[97]. In 2022, Wi-Fi vulnerabilities [12], [13] enabled code execution on the EVN manager. The attackers then bypassed the in-vehicle firewall to inject unfiltered UDP packets that were automatically translated into CAN frames. In 2023, Bfluetooth vulnerabilities allowed kernel-level execution, which was used to update IVN gateway firmware with malicious code. In 2024, LTE interface exploits enabled direct CAN injection from the EVN side, bypassing the gateway's intended filtering mechanisms. (~~**RQ1**~~, ~~**RQ2**~~, ~~**RQ5**~~)

These attacks collectively reveal several key observations highlighted above. A broader survey appears in Jing et al. [53], but we find that all but a few of their presented attack paths follow the above-described pattern.

*3) **Arm TrustZone TEE**:* Most processor vendors provide TEEs as security extensions to their processors. Arm's implementation is TrustZone [73], available on both Cortex-A and Cortex-M families. TrustZone partitions processor execution into two states: the *Secure World*, which hosts trusted services, and the *Normal World*, which runs the general-purpose OS and applications. The *Secure World* establishes the TEE for sensitive operations such as key management and message authentication, and it is entered from the *Normal World* via Secure Monitor Calls (SMCs). The TrustZone Protection Controller (TZPC) restricts peripheral access, while the TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA) enforce secure memory access over DRAM and SRAM, respectively. Together, they provide hardware-enforced isolation that grants the Secure World exclusive access to critical memory and devices, forming a robust TEE. This TEE underpins HTA functions; key management, cryptographic operations, and other HTA requirements [75]. The secure monitor (EL3) mediates communication between worlds: it handles SMCs, controls interrupt routing (allowing selected IRQs to be delivered to the Secure World OS), and can trap security-critical operations (e.g., control-register updates), effectively deprivileging both worlds' OSes from low-level mechanisms such as page-table management.

*4) **Automotive HSMs**:* Automotive HSMs are embedded co-processors that provide hardware-isolated execution for security services. They are specified in three variants, EVITA-*full*, *medium*, and *light*, by the EVITA project [106]. The *full* variant offers the strongest security capabilities required for EVN–IVN protection, while the *medium* and *light* variants primarily support in-vehicle communication security, often integrated into security-critical ECUs and actuators for message authentication. Current EVN–IVN platforms from major vendors integrate an EVITA-*full* class HSM at the IVN gateway [22], [37], [89]. These HSMs provide tamper resistance, immutable boot firmware, random number generation, key management, and optimized cryptographic acceleration—features essential for strong automotive security [76].

C. Intra-Kernel Isolation

The EVN manager relies on a complex OS (e.g., Automotive Grade Linux (AGL) [61]) to manage the EVN platform's extensive resources and diverse applications, leading to a large TCB and heightened vulnerability to bugs and exploits. Moreover, they commonly adopt a monolithic design, integrating the core kernel and device drivers within a single address space; thus, a flaw in any one component can compromise the entire OS, including the core kernel itself. To counter this risk, several works [27], [34], [50], [56], [60] propose intra-kernel isolation, which partitions a monolithic kernel into two domains: the *inner* and *outer* kernel. The inner kernel retains privileged control over security-critical resources, such as the memory management unit (MMU) and page tables [60]. To achieve this, some approaches [34], [56] leverages ARM64 features like `TxSZ`, a field in the `TCR` control register that

defines the virtual address length, to dynamically adjust each domain's address space and restrict memory access and execution privileges. Transitions into and out of the inner kernel occur via small *gates* that perform security checks and reconfigure address-space size and permissions, and this design has been formally analyzed with proven security guarantees [45].

III. Threat Model, Security Objectives, and Challenges

Problem Definition. Control messages ultimately drive vehicle actuation. In current deployments, control messages that go from the EVN to IVN ECUs are checked only at the IVN gateway using packet-level filters. This leaves a gap where EVN-originated control messages can satisfy these filters yet still violate OEM safety and security policies, because the IVN gateway has no visibility into the EVN manager runtime context or per-application provenance on the EVN side. On the EVN platform, applications and the OS that hosts them, along with the network stack and drivers, form the control message transmission path. However, either of these can be compromised; access control over message buffers over this transmission path is insufficient, and messages lack a concrete, authenticated binding to their true originating application. As a result, there is no practical mechanism to robustly enforce per-application sender policies on EVN-to-IVN control traffic.

Attacker Model. SECV focuses on software adversaries in the EVN domain that exploit the contextual gap defined earlier to cause unauthorized vehicle control. The attacker compromises the message transmission path on the EVN manager, aiming to arbitrarily inject, modify, or replay control messages that reach IVN ECUs via the IVN gateway while bypassing OEM policies. We categorize the adversaries as follows:

(a) Application-level attacker: The attacker runs arbitrary code in one or more EVN application processes in user space, but has not gained kernel privileges. The attacker can invoke standard EVN networking APIs to send control messages, but cannot directly modify kernel data structures, driver state, or peripheral registers. Under this capability, the attacker forges control messages at the application layer to spoof provenance: for example, by exploiting applications without transmission privileges to pose as authorized senders, or by implanting backdoors in authorized applications that emit attacker-chosen control messages under a legitimate application identity.

(b) OS-level attacker: In the stronger case, the attacker defeats the OS defenses of EVN manager (e.g., via privilege escalation or sandbox escape) and gains kernel-level control over the whole transmission path. With kernel privileges, including control over memory access, process management, and OS-level communication mechanisms, the attacker can *spoof provenance* by forging and injecting messages that appear to originate from privileged applications so that they pass IVN gateway static filtering, placing crafted frames into socket buffers, network stack queues, or communication peripheral transmit buffers; *tamper on path* by modifying or replaying control messages that applications have handed off to the OS through manipulation of kernel-space message buffers and transmission queues; and perform *driver/peripheral subversion* by compromising communication drivers or directly programming peripheral transmit buffers (e.g., CAN controller mailboxes) to place attacker-crafted frames on the physical bus and attempting to use DMA-capable peripherals to access memory regions used by trusted components in an effort to subvert SECV.

Assumptions. Physical attacks (e.g., electromagnetic or voltage fault injection [55]) that require specialized countermeasures beyond HTAs, side-channel, and other hardware attacks are out of scope of SECV. We assume an EVN platform is an ARM-based platform equipped with TrustZone as the TEE, and an IVN platform equipped with an automotive-grade HSM equivalent to EVITA-Full. EVN applications run in the Normal World. The TEE is trusted, and direct attacks on it are out of scope. The Normal World OS is untrusted, aside from an isolated minimal trusted component that handles core kernel functions, as detailed in § IV.

Security Objectives and Challenges. The primary objective of SECV is to prevent unauthorized vehicle control from the EVN domain, by establishing secure, authenticated message paths from EVN applications to the IVN gateway. To this end, through a trusted enforcement module on the EVN platform, SECV ensures that any control message delivered to the IVN gateway from the EVN manager is authenticated, integrity-protected, and policy-compliant before it leaves the EVN. Concretely, SECV provides the following three properties.

- **C1:** *Message authenticity:* A message is *authentic* only if it is bound to a legitimate EVN application identity. Messages emitted by backdoor applications or injected via a compromised OS must be detected and blocked before they reach the IVN gateway. The challenge is that the enforcement module runs in the TEE while all applications execute in the Normal World. Accordingly, SECV must both (i) verify the integrity/identity of the message-emitting application and (ii) systemically bind each message to that application, so the TEE-resident module can enforce per-application sender policies with accuracy.
- **C2:** *Message integrity:* The EVN side path, user-space buffers, kernel-space buffers, and driver accesses on control messages must be hardened so that, once issued, a control message cannot be altered or replayed at any subsequent stage. The challenge for SECV is to enforce buffer and path access control under an untrusted OS and device drivers, preventing untrusted components from spoofing provenance or tampering with in-flight messages to bypass the TEE enforcement module.
- **C3:** *Policy enforcement:* Per-application, least-privilege authorization is applied at send time, ensuring each message conforms to OEM policy (e.g., permitted IDs/topics, destinations, and rates) and denying any nonconforming transmission. To guarantee that no message is put on the wire without authorization, SECV revokes Normal World access to communication peripherals and binds device control to the Secure World. This, however, necessitates Secure World

driver support; a naïve port of full drivers would either be insecure or bloat the Secure World TCB.

IV. SECV DESIGN

A. Design Overview

As illustrated in Figure 9, SECV is a two-layered security framework that preserves the IVN gateway's static filtering while adding EVN side message authentication and policy enforcement, ensuring that only pre-authorized EVN-originated control messages reach the IVN gateway, by introducing two trusted enforcement components: the Secure Software Module for the EVN (`SSMe`), which runs in the TEE on the EVN platform, and the Secure Software Module for the IVN (`SSMi`), which operates alongside the HSM on the IVN gateway; together, they establish an end-to-end secure channel in which `SSMe` authenticates EVN-originated control messages, enforces per-application policy, and authorizes transmission by cryptographically signing messages, while `SSMi` verifies these signatures before the IVN gateway processes and forwards messages to IVN ECUs.

Since `SSMe` executes on the EVN platform, it can obtain sender-application identity (via the mechanisms described later) and bind each control message to its origin. Running in a trusted, isolated environment with exclusive control over communication peripherals, `SSMe` provides pre-authorized transmissions that the IVN gateway can safely rely on while retaining its existing static filters. To delegate these responsibilities to `SSMe` and establish a trusted end-to-end channel, the IVN gateway must verify `SSMe`'s isolation and integrity. SECV achieves this via boot-time verification and remote attestation of the EVN TEE (including `SSMe`), with the IVN gateway acting as verifier. SECV leverages secure boot and extends measured boot on the EVN platform to produce attestation evidence for EVN side trusted components. This evidence is conveyed to the IVN gateway and validated by its HSM against reference values and policies stored in secure HSM memory. Only upon successful validation does the IVN gateway permit secure channel provisioning; the detailed attestation and channel-setup flow appears in § IV-B1.

On the EVN side, SECV designates communication peripherals (e.g., CAN or SPI controllers) as choke points for transmission authorization, since they are the final control point for messages. It revokes Normal World access and binds these peripherals to the Secure World using TZPC and TZASC, so that `SSMe` must mediate all transmissions. SECV therefore restructures the message-transmission path. In conventional architectures, EVN applications submit messages to the Normal World OS, which copies them into kernel-space buffers and processes them through the network and driver stacks until drivers write them into peripheral Tx buffers—a path exposed to the attacks described in § II and § III. In contrast, SECV introduces *secure message buffers*: memory regions allocated and owned by `SSMe` in the Secure World and uniquely bound to an EVN application, while non-writable by untrusted Normal World components (including the OS). Each application writes control messages only into its assigned secure buffers. Rather than copying payloads across untrusted OS buffers, SECV passes buffer references as needed, preserving a stable binding between each buffer and its originating application.

Because the Normal World is denied peripheral access, drivers submit transmission requests to `SSMe` by referencing these secure buffers. `SSMe` uses the buffer addresses to attribute each request to an application and enforce per-application transmission policy before placing data on the wire. This design requires (i) a trusted conveyance of application identity to `SSMe` for correct buffer provisioning and provenance binding, and (ii) write-restricted buffers with non-remappable ownership under an untrusted Normal World OS. To realize these properties, SECV adopts intra-kernel isolation, partitioning the Normal World OS into a minimal, attested-and-trusted inner kernel (responsible for core kernel operations) and an untrusted outer kernel. The inner kernel manages memory and processes, and SECV relies on it to supply accurate process identity and enforce buffer access control within the Normal World; further details appear in § IV-B2.

Finally, restricting communication peripheral access to the Secure World requires SECV to provide Secure World drivers. A naïve port of full Normal World drivers would inflate the Secure World TCB and, given the prevalence of high-risk driver vulnerabilities [6], [17], [18], [20], [21], import substantial attack surface [44], [83], [84], [105]. Instead, SECV adopts a split-driver architecture: minimal driverlets in the Secure World implement only the authorization-critical control path and the minimal device operations necessary to place data on the wire, while the feature-rich Normal World drivers handle non-security-critical functionality. Driverlets cooperate with the Normal World drivers through a constrained, validated interface (e.g., parameter-checked commands and controlled shared buffers) so that transmission can proceed only after `SSMe` authorization, as *detailed* in § IV-B5.

B. Design Details

*1) **`SSMe`-to-IVN gateway Secure Channel Establishment**:* **IVN Gateway Attestation of the `SSMe`.** SECV mandates secure boot and measured boot anchored in hardware roots of trust on both sides, the HSM in the IVN gateway and the TrustZone-based platform in the EVN manager. To enable cross-domain trust, the OEM provisions asymmetric key pairs for both endpoints prior to deployment: the HSM retains its private key and the EVN manager 's public key in secure NVM, while the EVN manager stores its private key and the HSM's public key in hardware-protected storage (eFUSE/OTP) accessible only during early boot.

Secure EVN–IVN Channel Establishment. SECV establishes the `SSMe`–`SSMi` channel in three phases. *Phase 1 (bootstrap attestation):* initial EVN measurements (up to BL2) are conveyed under the provisioned asymmetric keys; after successful verification via the HSM, entropy is supplied to derive a symmetric root session key (`SKR`), which is also imported into the HSM. *Phase 2 (TEE attestation):* `SSMe` trans-

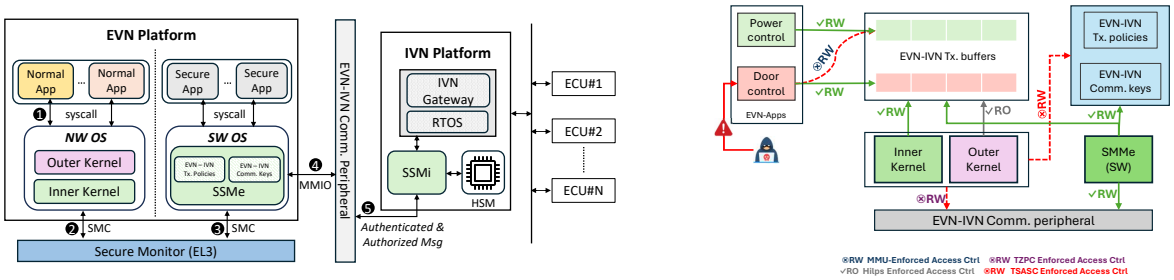


Fig. 2: An Overview of SECV Design and Security Enforcement for EVN-IVN Communications. Left: SECV's EVN-IVN security components; Right: SECV's regulation of EVN application control message transmission to the IVN.

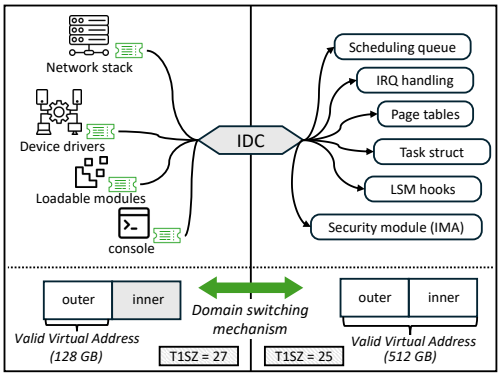


Fig. 3: Normal World OS partitioning into a minimal trusted inner kernel and untrusted outer kernel.

mits the final measurement set—including the Normal World OS—under SKR, and SSMi verifies it via the HSM. *Phase 3 (runtime channel):* upon success, SSMi delivers fresh entropy protected under SKR; SSMe seeds its PRNG and derives a runtime session key (SKC) for lightweight, ongoing protection of control-plane messages. An SKR-wrapped copy of SKC is imported into the HSM; SKR is then discarded, and all subsequent SSMe ↔SSMi traffic uses SKC.

This hybrid protocol completes EVN manager attestation and secure key provisioning while cleanly separating boot-time attestation keying (SKR) from runtime communication security (SKC).

*2) **The Minimal Trusted Normal World OS Inner Kernel**:* As stated earlier, SECV partitions the Normal World OS into a minimal trusted inner kernel and an untrusted outer kernel to achieve intra-kernel privilege isolation. SECV builds on and repurposes Hilps [34] to realize an isolated, trusted kernel component by dynamically reconfiguring the kernel's address-space view via the TxSZ field. During inner kernel execution, TxSZ is set to expand the translation range to cover both inner and outer kernel memory; during outer kernel execution, it is compressed so the outer kernel's translations exclude inner kernel regions, restricting access to its assigned range only. Figure 3 depicts an overview of how SECV partitions the kernel, and a sample of constituents of each kernel component.

The inner kernel comprises core kernel components and holds exclusive write access to page tables and control registers (e.g., page-table base registers TTBR). Switching between inner and outer kernel execution is performed via an *interface domain call (IDC)* gate [34], a defined instruction sequence that securely reconfigures the address-space view. Beyond default memory management, SECV entrusts the inner kernel with process management and scheduling, thereby controlling access to process control data structures by allocating the caches from which they are drawn in the inner kernel address region. It also moves scheduling code and process-lifecycle functions to the inner kernel region. Security-critical subsystems, including integrity measurement architecture (IMA) control logic, security-sensitive data, and security module (LSM) hook dispatch, are also placed in the protected region; the outer kernel may only issue service requests to these components. The inner kernel further protects interrupt/exception vector tables to prevent hijacking during interrupt handling.

Non-security-critical components, and those historically prone to vulnerabilities, such as device drivers, network stacks, and most process-service logic, remain in the outer kernel. Memory objects that the outer kernel must read but not modify (e.g., page-tables, scheduling queues or task_struct fields) are mapped in regions that are read-only to the outer kernel and read-write to the inner kernel, preserving necessary observability while maintaining write protection.

Note that SSMe may be invoked by either the inner or the outer kernel, and it cannot, by itself, distinguish which compartment initiated the world switch. Yet some SSMe requests are privileged and must originate only from the inner kernel; a compromised outer kernel could otherwise abuse them to subvert SECV. To prevent this, SECV leverages the EL3 secure monitor to mediate all world switches and block privileged SSMe invocations from the outer kernel. The monitor inspects TCR_EL1.T1SZ to identify the caller's compartment. Because only the inner kernel is permitted to set the distinguishing T1SZ values, this hardware-enforced field allows SECV to reliably detect and reject forged transitions from the untrusted outer kernel.

*3) **Guaranteeing Message Authenticity**:* After boot, EVN applications can send control messages to the IVN. Since applications may be dynamically loaded and are unauthenticated at boot, attackers may install malware, tamper with binaries, or forge communication policies to inject spoofed messages (see § II). SECV treats such messages as inauthentic and seeks to detect and prevent their transmission. SECV binds message

Key	Type	Purpose	Gen. By	Persist.	Storage
SKR	AES	Runtime SSMe/SSMi trust	HSM	Ephemeral	HSM / secure RAM
SKC	AES/SHA	SSMe ↔ SSMi comm.	SSMe (TEE)	Ephemeral	HSM / SSMe secure RAM
IVK	RSA/ECC	IMA signature verification	OEM	Permanent*	HSM sNVM / SSMe secure RAM
MEK	AES	Encrypt MIDs	OEM	Permanent*	HSM sNVM / SSMe secure RAM
HSM pair	RSA/ECC	Root of SSMi/SSMe trust	OEM	Permanent	Pub: eFUSE / Priv: HSM
EVN mgr pair	RSA/ECC	Root of SSMi/SSMe trust	OEM	Permanent	Pub: HSM / Priv: eFUSE

TABLE I: Keys used in SECV. *Ephemeral* keys are valid for a single boot; *Permanent** indicates OTA-updatable keys. All keys reside in HSM NVM or EVN secure storage and are accessible only to HSM or EVN trusted components.

authenticity to the integrity of the transmitting application and its associated transmission policy at load time. Concretely, it extends Linux IMA, requiring that application binaries and policy be signed prior to deployment. At load time, IMA verifies each file against a trusted key in its keyring (signature appraisal) and permits execution/use only on successful validation; failures are denied, preventing unmeasured or unsigned code and policies from issuing control messages.

As shown in Figure 4, at deployment SECV requires the OEM to encrypt the transmission policy (or policy set) with a symmetric Message Encryption Key (MEK) and bundle it with the application binary (❶, ❷). The bundle is then hashed (❸) and the hash digitally signed (❹) with a private key whose public counterpart is installed in the EVN platform's keyring. At load time, the inner kernel recomputes the bundle hash and verifies its integrity (❺, ❻) using the keyring public key, denying execution on failure (❽). Upon success (❼), the inner kernel unbundles the application, allocates a process identifier (PID), binds the PID to the encrypted policy, and forwards this policy and PID metadata to SSMe (❾). SSMe decrypts the policy with the MEK copy, which is stored in the HSM's secure non-volatile memory (sNVM) and can be retrieved at boot after the SSMe-SSMi secure channel establishment, and securely associates the permitted message identifiers (MID) with the process in a PID-MID mapping structure, keeping this mapping hidden from the Normal World. To preserve the binding across process creation, SECV hooks fork so that children inherit the parent's policy; if fork is followed by execve, the inner kernel notifies SSMe to remap permissions to the new binary and policy.

SECV hardens IMA by adding a keyring verification step to the Normal World boot sequence, preventing attackers from spoofing signature appraisal with a tampered keyring. Specifically, the OEM provisions an IMA Verification Key (IVK) used to sign the keyring; at boot, the inner kernel verifies the keyring against the IVK before enabling IMA policy and appraisal. The IVK public key may also be provisioned in the HSM's sNVM and retrieved at boot (similar to the MEK) to provide a hardware-anchored reference. Finally, unlike stock IMA, SECV requires that any changes to IMA measurement/appraisal policy require both root privileges and explicit authentication by SSMe using an OEM-specific secret. This dual control prevents even privileged attackers from disabling IMA or silently altering its policies.

*4) **Normal World Control Message Transmission**:*

Sender Application Provenance. To achieve accurate per-application policy enforcement, SECV must reliably associate each outbound control message with its sender application.

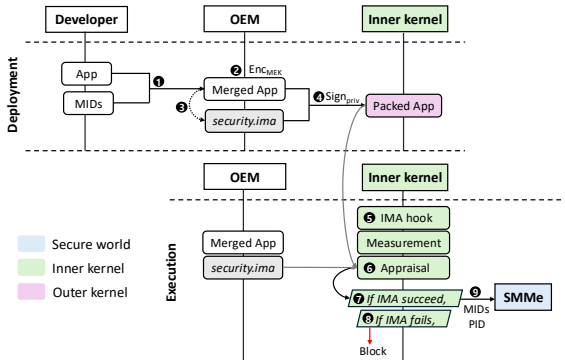


Fig. 4: Deployment and Execution SECV's IMA Flow.

One approach is to have applications or the OS attach a PID to control messages, but this is insecure since untrusted components could forge such metadata. Another approach is to assign provenance tracking to the inner kernel, but this would require it to manage the entire message path, significantly expanding its TCB.

Instead, SECV enforces provenance using per-process secure message buffers. Upon an inner kernel request that presents a process PID, the SSMe allocates a dedicated shared-memory buffer pool for that process, records the pool's physical base address in the PID–MID map structure (§ IV-B3), and returns the pool address to the inner kernel. Control messages for that process are written into buffers carved from this pool. When the SSMe (driverlet § IV-B5) receives a transmission request, it identifies the sender by resolving the buffer's address via the PID–MID map structure, eliminating any dependence on untrusted metadata.

Guaranteeing Message Integrity. To protect control messages from compromise, especially as they traverse the outer kernel path, SECV relies on the inner kernel to enforce access control over the submitted message buffers. When the inner kernel receives a pool's physical base from the SSMe, it maps the pool into the process address space as read–write, maps it read–write for the inner kernel, and maps it read-only for the outer kernel. Because the network stack and Normal World peripheral drivers reside in the outer kernel, they cannot write to these buffers and therefore cannot tamper with in-flight control messages.

Buffer pool management is securely, jointly handled by the inner kernel and SSMe. For each buffer pool, they maintain shared metadata in the form of a bitmap and a buffer-size field. The inner kernel allocates a message buffer by setting the next available bit in the bitmap and returning the corresponding buffer address. When SSMe receives that buffer in a transmit request and successfully sends the contained control message,

it frees the buffer by clearing the corresponding bit in the bitmap.

Another concern arises from the Normal World OS's use of APIs like `copy_to_user, put_user` [100], and related functions that write to user space. Although the transmit buffer pool is mapped read-only to the outer kernel, these APIs may still be exploited. For instance, a malicious outer kernel could use an unrelated system call to trigger one of these APIs and overflow a user-supplied buffer into the memory region backing the transmit pool. This violates isolation, allowing the outer kernel to tamper with message post-submission. To address this, SECV hooks the `copy-to-user` family of APIs and enforces a constraint: writes into `SSMe`-mapped shared-memory regions are permitted only when performed by the inner kernel. All other attempts are denied, ensuring strict control over Secure World –to–Normal World data flow and preserving the integrity of submitted message buffers.

The Message Transmission Path. Application processes transmit control messages typically through the network stack using sockets and `send` or `write` system calls, and sometimes, although rarely, directly to peripheral drivers via the `IOCTL` interface. When a process attempts to transmit a control message, the inner kernel intercepts the system call. It allocates a buffer from the process's pool, copies the message into it, and forwards the buffer's address to the outer kernel's network stack. The outer kernel uses the standard `sk_buff` structure [101] to carry the control message through the networking layers to the Normal World OS peripheral driver.

However, as the message traverses the network stack, certain operations may require write access to the message buffer. This presents a challenge for SECV, which must switch between the inner and outer kernels to verify and perform the intended write on the outer kernel's behalf. These transitions incur performance overheads and complicate enforcement. To address this challenge, we examine the transmission path of the widely used CAN protocol. We focus on CAN in our design and evaluation of SECV because CAN drivers are tightly integrated with the kernel and typically rely on large, monolithic frameworks for message handling, error processing, and bus arbitration, making it highly complex to partition the logic between the Normal World driver and the Secure World driverlet in `SSMe`. We note that Automotive Ethernet is rapidly gaining adoption, particularly in Tesla vehicles and increasingly across the industry [47], and we detail in § VII how our CAN-centric design can be extended to support Automotive Ethernet with minimal modifications.

We observe that most modern EVN applications transmit via SocketCAN, using the standard socket API. Along the network path, there are a few sites where the submitted CAN frame is mutated (e.g., protocol headers/flags). These mutations are minimal and typically derived from fields already set by the application; accordingly, the stack can temporarily switch to the inner kernel, which validates the intended changes and performs them on the stack's behalf. Crucially, we find no need to reallocate or deep-copy the frame as it traverses the stack, except for a few instances in SPI-wrapped CAN. Although

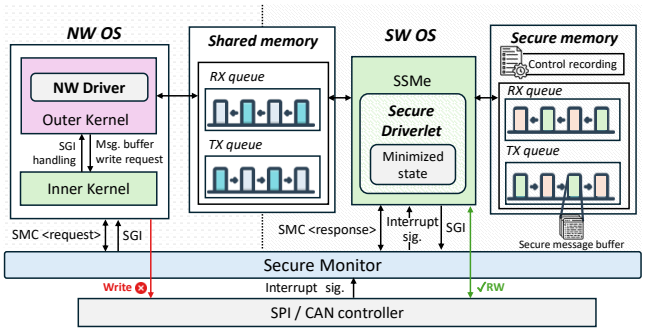


Fig. 5: SECV driver splitting setup, using shared buffers, SMCs, and SGIs for communication.

`sk_buff` normally co-locates metadata and payload in a single allocation, it also supports externally allocated payloads via `data`. This enables straightforward adoption of SECV 's message-buffer design by setting `data` to the secure message buffer and letting the stack operate over that buffer under inner kernel mediation.

5) **SECV *Driver Splitting*:** In line with SECV 's goal of preventing unauthorized transmissions, SECV revokes Normal World 's direct access to peripheral drivers, ensuring that a compromised Normal World component cannot bypass `SSMe` and inject unauthorized control messages. As described in § IV, SECV achieves this by adopting a split-driver model that partitions existing peripheral drivers into an Normal World component and a minimal Secure World driverlet.

This model presents a further challenge: EVN–IVN communication peripherals (e.g., CAN, UART, SPI) rely heavily on MMIO. Because SECV confines peripheral access to the Secure World, these MMIO regions are accessible only there, requiring all interactions to be mediated by `SSMe`. However, existing drivers tightly interleave MMIO operations with complex state management, error handling, and kernel-dependent logic, leaving no obvious boundary for safe partitioning. The difficulty is compounded by low-level I/O patterns: drivers frequently manipulate peripheral registers byte-by-byte, even when processing entire frames, with the relevant logic dispersed across the codebase.

Splitting Points and Coordinated Control. We observe that transmission and reception paths dominate peripheral interaction and are executed most frequently. Accordingly, SECV relocates the peripheral-access logic specific to message Tx/Rx into `SSMe`, decoupled from the original driver structure. In the Normal World drivers, this logic is replaced with explicit SMC requests to Secure World *driverlets*. Driverlets implement simplified control flow and minimal state to track transmission, reception, and essential peripheral state.

For the remaining logic, the Normal World driver issues access requests to the driverlet as needed. SECV treats read requests as benign, but write requests as potentially malicious. In practice, control registers are typically well-identified; driverlets are protocol-aware and encode the permitted peripheral states and transitions. Consequently, they can whitelist

expected control sequences and flag any Normal World write sequences that deviate from the known protocol. Prior work demonstrates similar approaches, pre-recording peripheral accesses and replaying or validating them at runtime [44], [105]. This preserves Secure World ownership of device registers while allowing the outer driver to progress using validated, least-privilege state.

Message Transmission and Reception. For message Tx/Rx, SECV adopts a zero-copy design that shares buffer addresses between Normal World drivers and `SSMe`. On the Tx path, this is already provided by per-application pools and secure message buffers. On the Rx path, SECV allocates a common shared memory pool for all incoming IVN frames; while receive buffers are not per-application by default, they can be logically partitioned to prevent EVN applications from reading messages they are not authorized to access. SECV coordinates exchange via software shared queues. For Tx, the NW driver enqueues a buffer address (from the network stack) into the Tx queue and issues an SMC; the driverlet validates the request and, if admissible, programs the device to transmit directly from the referenced buffer. For Rx, the driverlet writes incoming frames into a buffer from the shared pool, enqueues its address into the Rx queue, and signals the NW driver with an SGI. This scheme preserves zero-copy semantics while keeping `SSMe` in the authorization loop for both directions.

Real-time EVN–IVN communication is heavily interrupt-driven. Since `SSMe` controls all peripherals, SECV routes all hardware interrupts to the driverlet, which then notifies the NW driver via SGIs. A naïve scheme, one SMC call per transmission and one SGI per received or completed frame, incurs frequent world switches, leading to high latency and throughput overheads. To address this, SECV adopts a NAPI-inspired polling and batching model in the split-driver path. Instead of handling every peripheral interrupt individually, the driverlet disables them after the first and polls interrupt flags to process multiple Tx/Rx events in a single invocation. This approach significantly reduces world switches during burst traffic, e.g., when the FlexCAN controller raises rapid successive interrupts-avoiding severe reception frame loss. SECV also amortizes transmission by batching. Finally, because the number of available SGIs is limited, while peripheral controllers typically raise distinct interrupt lines, SECV uses a single SGI with a shared-memory bitmap for demultiplexing.

*6) **Policy Enforcement and Actual Message Transmission**:* For each control message dequeued from the Tx queue, the driverlet retrieves the associated policy using the buffer address, which maps to a PID-MID map entry identifying the sender. It verifies whether the application is authorized to send the message, by checking the ID (or a destination IP/Port in case of Ethernet) of the transmission request against the MIDs in the PID-MID map entry. If authorized, it signs the message with SKC and writes it on the peripheral buffer for transmission, and frees the buffer by unsetting the corresponding bit in the bitmap. Any mismatch indicates a replay or fault.

*7) **IVN Message Reception by and `SSMi` Role**:* By default, SECV assumes the IVN is benign and focuses on blocking EVN-originated attacks from reaching it. For EVN-bound traffic, however, `SSMi` signs or encrypts messages at the IVN gateway, and `SSMe` decrypts them with the SKC, enqueues them in the Rx queue shared with the NW driver, and raises an SGI so the driver forwards them to the NW stack. SECV can also support application-specific reception by establishing reverse paths between `SSMe` and the NW driver, analogous to the transmit paths.

V. IMPLEMENTATION

We implement a SECV prototype on an ARMv8-A platform, building the `SSMe` atop OP-TEE v4.0 and extending the ARM Trusted Firmware v2.10 secure monitor, with Linux 6.6.52 in the Normal World. Our implementation focuses on the EVN side—with only minor IVN modifications—and includes the `SSMe` and kernel isolation mechanisms, CAN driver changes for the NXP S32G3 platform, a generalizable SPI-wrapped CAN driver using the MCP2515 CAN HAT, and extensions to Linux's IMA, MMU, process and task management, security modules, and CAN network stack.

A. Intra-Kernel Isolation

To partition the kernel into inner and outer, we build on Hilps [34]'s `IDC` and initial page-table setup, which split the kernel virtual address space into two compartments. We hook the `set_pXX()` family so that page table updates are verified and applied in the inner kernel, and extend the linker script to place security-critical code and data in the inner region. In particular, we map most of the `SCHED_TEXT`, `IRQENTRY_TEXT`, and other sensitive sections, as well as process/task-management such as `task_alloc`, `copy_process`, and `sched`, into the inner kernel. For non-prefixed functions and globals, we introduce sections prefixed with `__secv` and collect them into a dedicated range in the linker script. Finally, we hook `copy-to-user` APIs and related helpers with checks that ensure only inner kernel can modify message buffer ranges.

For the LSM framework, we register SECV-specific hooks with `security_add_hooks` in the outer kernel, but implement their handlers in the inner kernel. Each LSM hook in the outer kernel therefore points to a stub that marshals arguments and invokes the `IDC`, where the actual handler runs. For process-management hooks such as `bprm_check_security`, the stub forwards the `linux_binprm` context to the inner kernel, which cooperates with SECV-extended IMA to establish the PID–MID binding and trigger Secure World updates before `execve` completes.

B. Implementing the `SSMe`

We implement the `SSMe` as an OP-TEE driver consisting of multiple driverlets and a companion library. We define SMC function IDs to service Normal World OS requests. The core library provides a page-sized message-buffer pool carved from a reserved shared-memory region and maintains the PID–MID map as a hash map keyed by PID hashes. For cryptography,

secure storage, and other HTA-related services, we bootstrap OP-TEE's `fTPM` [77].

Driverlets are derived from Normal World driver logic. We define minimal state to track only what is required for Tx/Rx and protocol compliance, and implement functions as pared-down adaptations of Normal World routines or clean reimplementations informed by driver behavior. On the transmit path, we target `__start_xmit()`, which consumes an `sk_buff` and typically contains the MMIO sequences that effect CAN-frame transmission in dedicated CAN drivers (e.g., `flexcan`, `rcar`). For SPI-attached CAN devices, this remains a convenient post–network-stack interception point, and we additionally inspect `__transfer_one()` in the relevant SPI driver. Where device manuals are explicit, we further minimize logic and move only the necessary MMIO accesses, akin to record-and-replay.

On the receive path, we study the polling/IRQ handlers (e.g., `__irq_do_()`) that produce an `sk_buff` and extract the minimal code needed to fetch frames from the device, with the remaining path proceeding in the Normal World via SMCs. Finally, we implement interrupt handlers, either as minimal counterparts of their Normal World versions or as coordination routines that work with Normal World drivers to manage peripheral state.

C. Normal World Drivers

We modify existing normal world drivers to ensure they are in sync with the secure world driverlets. We replace peripheral access sequences that result in message transmission with SMC code that enqueues frames to the shared queue and makes SMC requests to the `SSMe` driverlets. As explained above, we locate and transform the `no_start_xmit()` routines.

For interrupt handlers, e.g., `flexcan_irq_mailbox` of the FlexCAN driver, since the driverlets already define minimal versions of their own, we implement new functions that coordinate with the driverlets to handle the driver state. These handle SGIs forwarded by the `SSMe` instead of the original interrupt. We also hook the `readl`/`writel` helpers in the CAN and SPI drivers, replacing their MMIO accesses with SMC requests. The driverlets interpret the accessed addresses according to a known static MMIO protocol, which allows them to block malicious peripheral accesses.

D. Modifying The CAN Network Stack

Having analyzed the CAN network stack, we identify and modify critical sites relevant to SECV. Specifically, we adjust the `__sendmsg` path (e.g., `raw_sendmsg()`, `bcm_sendmsg()`) where `send`/`write` copy the control message into kernel space. SECV ensures that the allocated `sk_buff` is backed by the sender application's secure message buffer and that the payload copy occurs while temporarily switching to the inner kernel. Subsequent header-touch points along the path, before invoking `ndo_start_xmit()`, can be reasoned about and safely performed via inner kernel mediation.

Metric	Baseline (ms)	SECV (ms)	Overhead (×)
fork + execve	1.24265	1.53795	1.24
fork + exit	0.47311	0.61307	1.30
write	0.00054	0.00054	1.0
open/close	0.00703	0.00710	1.01
stat	0.00390	0.00376	0.97
fstat	0.00087	0.00088	1.01
send	0.00306	0.00417	1.36
recv	0.01085	0.01077	0.99
mmap	0.05100	0.06093	1.19
Geomean			1.11

TABLE II: SECV's LMBench results and overhead for affected system calls.

VI. EVALUATION

In this section, we evaluate SECV 's security guarantees and runtime performance on an S32G3 board with eight Cortex-A53 application processors (hosting the EVN manager), four Cortex-M7 MCUs (hosting the IVN gateway), an automotive-grade HSE, and peripherals including FlexCAN, LINFlex, SPI, UART, and automotive Ethernet. Although AP–MCU communication can use any of these links, we primarily use FlexCAN, with SPI/UART only for cases such as boot-time trust setup, consistent with § II. We measure SECV 's impact on EVN–IVN throughput and latency, EVN application performance during control-message transmission, and IVN gateway CPU load due to `SSMi`, and additionally evaluate its effect on system boot time, application load time, and overall security guarantees.

A. Performance Evaluation and Resource Utilization

*1) **Boot Time Overhead**:* We first evaluate SECV's impact on platform boot time, focusing on TEE firmware attestation and HSM trust establishment. Establishing trust between the HSM and `SSMe` adds only $5.83ms$ on average—less than 1% of total boot time. This overhead mainly arises from communication latency, firmware hash exchange, and the cryptographic operations needed to secure the protocol (encryption, decryption, and shared key generation).

*2) **Application Load Time and Affected Systemcall Overhead**:* Next, we evaluate SECV's impact on application load time to capture the cost of dynamic application attestation and binding EVN applications to the `SSMe`. To this process, SECV adds binary/policy verification and PID–MID map setup in `SSMe`, which increases load latency by $23\times$ (from $0.0256\,\mathrm{ms}$ to $0.593\,\mathrm{ms}$). Analysis of this overheaded shows two roughly equal contributors: (i) application/policy verification and (ii) the SMC to create the PID–MID entry. For applications without transmit permissions, SECV skips PID–MID setup; for others, setup can be deferred until the first send to amortize it with initial secure buffer–pool allocation. Because verification entails file hashing, we evaluated sensitivity to binary+policy size and MID count by averaging overhead incurred over bundles from $8\,\mathrm{KB}$ to $200\,\mathrm{MB}$; the overhead remained in the same range across sizes. Since this overhead is incurred only once during initial application loading, it does not affect subsequent communication latencies.

We measure the overhead that SECV introduces to the `fork` and `exec` system-call families using LMBench [63],

with missing calls measured manually. Table II reports these results, along with manually measured `recv`, `send`, and `mmap`. Recall that `fork` invokes the `SSMe` to create a PID–policy mapping for the child, while `exec` triggers IMA measurement and `SSMe` setup similar to application loading; because `exec` typically follows `fork`, we report combined overhead for `fork+execve`. We also evaluate the overhead of `write` and `send` used for EVN–IVN communication, which incur inner–outer kernel transitions and additional checks in `put_user`, `copy_to_user`, and related APIs. Each call is executed 100 times and we report the mean. The first `send` or `write` from a process includes a one-time SMC for transmission–pool initialization, causing over $3\times$ overhead; because this occurs only once, we exclude it from the averages.

*3) **Communication Throughput and Latency**:* Next, we assess the impact of SECV on EVN–IVN communication in terms of throughput, latency, and IVN gateway load. We begin by isolating SECV 's non-cryptographic overhead using microbenchmarks that measure round-trip time (RTT) and throughput for control messages of varying sizes between EVN manager and IVN gateway. We use NXP's GoldVIP SAE J2284 [5]-inspired EVN manager-IVN gateway communication benchmark that emulates authentic EVN-IVN traffic. Under burst transmission conditions (0ms inter-message gap), SECV introduces a throughput and latency overhead ranging from 1.6% and 1.7% for 64-byte messages to 11.1% and 0.6% for 8-byte messages, respectively, in our FlexCAN setup. Smaller messages deplete transmit buffers more rapidly, triggering more frequent world switches and reducing the time budget for SECV to process messages. Consequently, using a long Tx queue for such messages significantly hurts throughput, latency, and leads to a high message loss. In fact, in our experiments, we notice that for 8-byte messages, any Tx queue longer than 1 message at a time leads to a message loss of more than 20%. For such messages, SECV proposes treating them as a priority, and transmitting them to the `SSMe` immediately without queuing. We notice that this significantly improves throughput, latency, and reduces message loss for such small messages. However, care must be taken to ensure attackers don't use such messages to launch a DoS attack on SECV. One way SECV can mitigate such attacks is by incorporating rate limiting in `SSMe`'s filtering. Fortunately, `SSMe` is designed with flexibility in mind, allowing for OEMs to adopt any advanced filtering methods before EVN-IVN transmission. With this design, we observe a mere 1.5% additional average frame loss for small messages under burst reception. We also observed that longer Tx queues benefit large-size messages (32-64 bytes), and believe a slow-fast hybrid design of these queues can further improve overall communication performance. We leave this for future work. Notice that in some cases, the throughput and latency of SECV are better than the baseline's. This is attributed to the shorter, optimized message offloading paths SECV uses in its split-driver design, especially where most message handling logic is replaced by deterministic paths due to static recording. Consequently, except for the 8-byte small messages, SECV 's overhead decreases with larger message sizes, as the peripheral takes longer to transmit each message, giving SECV sufficient headroom to manage buffers without overruns. At the IVN gateway, incoming messages are first received and inspected by `SSMi`, adding a moderate average CPU load of 2%. Under less intensive conditions (inter-message gaps $\geq$ 1ms), SECV introduces negligible overhead in all metrics, as shown in Figure 6. This is expected, since the TEE world switch completes well within the idle gap, allowing SECV to handle communication without delay. Given that CAN messages typically tolerate up to $10ms$ latency, SECV is well-suited for real-world deployments.

Component	LoC
HILPS	Original: $0.19k$, SECV: $2k$
`SSMe`	$2.7k$
Secure Monitor (EL3)	$1.2k$

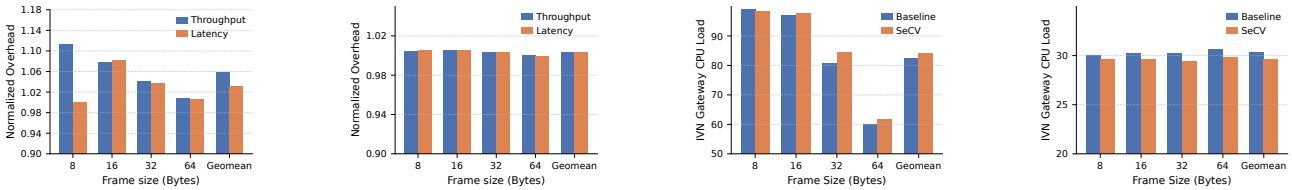
TABLE III: SECV's TCB size in terms of lines of code (LoC).

To evaluate SECV under realistic conditions, we replayed combined real-world CAN traces from CAN-MIRGU [78] and OTIDS [59]. Analysis of these datasets revealed no inter-message gaps below 1 ms, suggesting that the burst periods where SECV exhibits higher overhead are rare in practice. In our experiments, we observed no measurable throughput degradation or frame loss. Although SECV still incurred a small per-message latency overhead, all messages were delivered within their available time windows without overruns.

With encryption and/or authentication enabled, we evaluate SECV using HMAC on 32-byte messages with 32-byte MAC tags under a realistic 1ms inter-message gap. We simulate synthetic control messages from the EVN manager to the IVN gateway, each of which triggers an authenticated response, and run this exchange continuously for one minute. Throughput remains unchanged, but cryptographic processing adds an average round-trip latency of $181.26\,\mu$s, primarily due to MAC generation in the TEE and verification by `SSMi` via the HSM. At the IVN gateway, CPU load increases by 8.61%, which remains moderate because `SSMi` offloads expensive operations—especially response MAC generation—to the HSM. Using AES-GCM with a 12-byte nonce and an 8-byte tag for the same 32-byte messages, SECV incurs a higher average latency overhead of 246.79μs and a CPU-load increase of 13.99%.

For memory overhead, SECV reserves a fixed 30MB region at the EVN manager, from which it allocates all message channels and NW–EVN shared queues, avoiding dynamic allocation and ensuring predictable usage. SECV also reduces energy consumption by relying on peripheral polling instead of interrupts, which is particularly effective during message bursts and imposes only negligible CPU load on the IVN gateway. Consequently, SECV provides strong security guarantees without incurring excessive resource utilization.

*4) **Trusted Computing Base (TCB)**:* To reduce the risk of bugs and vulnerabilities in its core security components, SECV maintains a small TCB, summarized in Table III.



(a) Message throughput and latency (Transmission Gap $= 0ms$). (b) Message throughput and latency (Transmission Gap $= 1ms$). (c) IVN gateway CPU utilization (Transmission Gap $= 0ms$). (d) IVN gateway CPU utilization (Transmission Gap $= 1ms$).

Fig. 6: Combined performance impact of SECV under burst traffic.

B. Security Evaluation

We evaluate the security guarantees provided by SECV against the threats described in § III, and qualitatively explain its security guarantees against numerous attackers.

*1) **Application-level Spoofed Provenance**:* Attackers may install backdoors or alter applications, but such binaries fail the inner kernel's load-time integrity verification and cannot execute. Note that for an application to be verified, a cryptographic hash of its bundle (app binary + encrypted MIDs) must have been computed and signed by the OEM's private key, securely maintained remotely. Its digital signature must have been installed along with the bundle as outlined in § IV-B3. Verification encompasses recomputing the bundle hash and validating it against the public key counterpart of the OEM's private key (the keyring § IV-B3). Recall that SECV hardens this procedure by requiring further verification of the keyring in the Secure World, preventing attackers from subverting this process. Because transmission permissions are encrypted and bundled with the binary, and the bundle is signed, attackers cannot tamper with policies to bypass SECV 's authenticity guarantees. Likewise, exploiting a vulnerability in an installed application does not permit arbitrary control transmissions: `SSMe` enforces per-application permissions at send time, and nonconforming messages are denied and dropped. Even if a legitimate application is exploited to compromise the outer kernel, it cannot transmit control messages beyond its policy. The outer kernel is not authorized to allocate secure message buffers, nor can it spoof MIDs to `SSMe`, because only the inner kernel may relay PID–MID bindings to `SSMe`. It also cannot tamper with other processes' buffers: those pools are write-protected from the outer kernel and uniquely bound per process. Consequently, any message emitted by the compromised application is still recognized under its original identity and checked against its permitted MID set; attempts to transmit outside that set are denied by `SSMe`.

A more sophisticated threat involves hijacking an already-permissioned application to misuse its privileges and transmit unauthorized control messages. These may be addressed through rate-limiting strategies. Each application is allocated a bounded message buffer; abuse leads to buffer exhaustion and a subsequent lockout. To thwart timing-based inference attacks, each application may be assigned a randomized, per-session transmission rate, negotiated between the `SSMi` and `SSMe` during boot or application load.

*2) **Privilege Escalation Attacks**:*

Spoofing Provenance. An attacker who compromises the outer kernel may attempt to forge and inject control messages via the network stack or Normal World driver queues, attempting to impersonate an authorized application. However, such payloads reside in inauthentic buffers not recognized by `SSMe` 's per-process pool and thus do not resolve in the PID–MID map structure; the permission lookup fails, and the messages are denied and dropped.

Tamper/Injection on Path. Similarly, an attacker in the outer kernel may attempt to modify control messages submitted by authorized applications, or simply record and replay secure message buffer addresses. However, message buffer management is shared by both the inner kernel and `SSMe`. `SSMe` treats a message buffer whose bit is unset in the bitmap as carrying an invalid message and simply drops it.

Driver/Peripheral Subversion. An attacker cannot program peripheral Tx buffers as Normal World access is completely revoked by `SSMe`. Even the access requests the NW driver makes to the Secure World driverlets are strictly checked, especially if they are write requests to control registers. Interestingly, an attacker may compromise a Normal World driver to perform DMA attacks on the `SSMe`. On a system with a resource controller, such as XRDC on our evaluation board, peripherals can be constrained on the memory ranges they can access, thwarting such attacks successfully. On other systems with the SMMU, this can be placed in the control of the inner kernel and then leveraged to prevent such attacks. In the absence of a system resource controller or SMMU, static carve-outs of memory may be pre-assigned as in [105].

Compromised Firmware Attacks. Attempts to compromise the EVN manager OS or the IVN gateway firmware trigger boot-time attestation failures, resulting in either a halted boot process or blocked communication between the EVN and IVN domains. While such attempts may induce a denial-of-service (DoS) condition, they nonetheless prevent unauthorized control over safety-critical IVN components—a deliberate safety-preserving tradeoff in automotive security design.

Exploiting TEE Vulnerabilities. Advanced adversaries may attempt to exploit vulnerabilities in the Secure World OS (e.g., OP-TEE [14], [16]) to compromise `SSMe` itself. While such attacks lie out of the scope of this work, we outline several hardening options. For instance, `SSMe` can be implemented as a separate Trusted Application (TA) and isolated using XRDC, as demonstrated by ReZone [32]. Alternatively, `SSMe` can be executed on a dedicated, pinned CPU core with exclusive

peripheral access and shared memory communication, following designs proposed in Sanctuary [31]. XRDC isolation can further strengthen these setups by enforcing hardware-backed separation across trust domains.

VII. Discussion

A. SECV *for Automotive Ethernet & Protocol Translation*

We investigated whether SECV 's CAN-based design can generalize to Automotive Ethernet, whose adoption is steadily growing among major vendors (**AE5** in § II, [47]). However, the Ethernet stack is substantially more complex than CAN: the `sk_buff` structure is repeatedly reallocated and modified across layers (e.g., header insertion and flag updates), making frequent inner–outer kernel switches impractical, while pulling the entire stack into the inner kernel would bloat its TCB and import known vulnerabilities [7], [11], [15]. In principle, SECV could still support Ethernet by performing inner–outer transitions on each legitimate `sk_buff` update, but this would introduce prohibitive overhead and engineering complexity, motivating an alternative design for Ethernet and CAN–Ethernet protocol translation. Instead, we propose a kernel bypass approach [40], [42] that links applications to the `SSMe` via user-space network stacks and a modified NIC driver, which fits SECV 's per-application control model. The inner kernel maps isolated memory pools into each application's address space, user-space stacks allocate Ethernet packets from these pools, and the `SSMe` verifies packet buffers—as in our CAN design—before authorizing transmission to the NIC. With kernel bypass, SECV can thus be efficiently extended to Automotive Ethernet.

B. SECV *with Virtualization & Containerization*

Virtualization and containerization are increasingly used in connected vehicles to reduce ECU count and improve isolation. In such environments, SECV must still uphold its security guarantees, with communication peripherals mapped to the Secure World via hypervisor mechanisms such as nested paging. The main challenge is correctly associating applications and their virtualized network endpoints with IVN control messages. To address this, SECV can build on secure hypervisor modules such as those in [43], extended to support SECV and generalized beyond IVI domains.

C. Related Works

Secure Peripheral Access and Driver Splitting. Several works [38], [44], [62], [83], [84], [105] explore TEE-mediated secure peripheral access. Some, such as TEEFilter [83], manually split drivers at goal-specific boundaries (as SECV does), while others, e.g., Guo et al. [44], Liu et al. [62] and Wang et al. [105], record and replay driver interactions to synthesize minimal secure paths. To ensure availability in CPS settings, RT-TEE [105] adopts a split-driver design with debloated Secure World driverlets for secure tasks and full sandboxed drivers for nonsecure tasks, debloated by recording MMIO/interrupt interactions and replaying only the necessary subset. Recognizing that CPS peripherals (e.g., sensors and actuators) share buses, RT-TEE also enforces spatial and temporal isolation of device access using a layered bus scheduler.

SECV adopts a split-driver architecture akin to RT-TEE but uses manual partitioning narrowly focused on control-message Tx/Rx paths. Unlike RT-TEE's setting, where sensors/actuators commonly share a bus, automotive deployments typically place communication peripherals like CAN (and often SPI) on dedicated physical buses. This environmental setting difference further highlights the design goals and driver splitting designs between SECV and RT-TEE. Nonetheless, SECV's security remains intact even on prototype platforms with multiplexed, multi-purpose boards: SECV's driverlet can leverage peripheral selectors (e.g., `chip-select`) to identify the targeted device in each Normal World driver request and apply protocol-aware checks to block malicious transmissions. However, in shared-bus scenarios, SECV does not provide the availability guarantees pursued by RT-TEE. The inner kernel can detect availability degradation (e.g., contention/starvation), but strong availability requires specialized arbitration like RT-TEE's and is out of scope for SECV. We believe SECV could benefit from RT-TEE's layered scheduler, although not trivially, to provide availability guarantees, especially for high-priority control-message Tx/Rx, and from its template-based driverlet derivation to streamline driver splitting.

StrongBox [39] also relies on Normal World drivers (e.g., secure GPU use) and protects the data of trusted applications via a request–response interface with the Secure World, without Secure World driver mediation when untrusted apps use the GPU. By contrast, SECV mediates *all* peripheral control end-to-end: there are no windows in which a Normal World driver can transmit without Secure World driverlets' intervention. Every MMIO operation flows through protocol-aware Secure World driverlets in a request–response workflow that monitors, validates, and gates device access.

Although TEEFilter's driver splitting may appear similar to SECV, especially in the choice of splitting points, SECV faces a far broader MMIO surface than TEEFilter's NIC case (which touches relatively few Ethernet registers) and complex MMIO access-dependent logic, which largely influences driverlet design. Consequently, SECV driverlets make more autonomous decisions during peripheral access and handle message-related interrupts largely independent of the Normal World drivers.

Finally, manual driver splitting, as in SECV, does require engineering effort. That said, SECV offers a vendor-agnostic abstraction readily adaptable across CAN, SPI, and Ethernet. In practice, adoption chiefly entails: (i) registering device interrupts with Secure-World driverlets (given the IRQ number, add it to SECV 's table), (ii) defining a minimal driverlet state (typically vendor-specific but small; future work can mirror the Normal-World `sk_buff` abstraction), and (iii) refactoring key Normal-World routines (`xmit`, `recv`, IRQ handlers) into SECV 's request–response interface. To gauge portability, we examined Linux CAN and SPI-CAN drivers (e.g., Rockchip, R-Car, Kvaser, FlexCAN, `slcan`, `mscan`, `mcp2515`, `hi311x`). In all cases, the message paths (`xmit`/`recv`/`irq`) align well with SECV 's model; the main

divergence is vendor-specific interrupt handling. Here, SECV 's protocol-aware driverlets, and where helpful, RT-TEE–style template driverlets can bridge differences. For Ethernet, we expect an even simpler path: StrongBox-style [39] mediation can be adapted, potentially combined with kernel-bypass, as previously discussed.

EVN Attacks and Solutions. Several works demonstrate practical EVN-domain attacks, including full chains that start in the EVN, traverse the IVN gateway, and compromise IVN ECUs. Jeong et al. [52] report competition-style attacks on IVI systems, typically part of the EVN manager, with some reaching the IVN gateway—the threat surface SECV aims to protect. Other practitioners [57], [64], [95] show even more severe attacks that target broader components of the EVN manager. As discussed in Section II, these findings strongly motivated the development of SECV. More recently, Jing et al. [53] presented attack paths affecting vehicles from multiple vendors, further underscoring the severity and real-world relevance of the problem SECV addresses.

In contrast, relatively few works focus on defense. Harness [43] proposes a framework that restricts IVN gateway access from compromised IVI systems by defining a minimal trusted domain via a hypervisor. Tailored to Android Automotive OS (AAOS), it shares SECV's goal of securing paths from external sources to IVN ECUs, but we observed significantly higher communication overhead than SECV, likely due to the lack of hardware-backed cryptographic acceleration on the IVN side and evaluation on a non-autonomous platform. Moreover, while Harness establishes trust between the enclave and IVN ECUs (similar to SECV), it focuses only on GUI inputs from the touchscreen and relies on a hypervisor-based EVN architecture, which entails a larger TCB and greater attack surface than TEE-based designs.

Hardware Trust Anchors in Automotive Systems. Plappert et al. [75], [76] survey the use of hardware trust anchors (HTAs) in automotive security, including EVITA [106] HSMs, used on our IVN gateway, and TEEs. However, their treatment of combined HTA designs is brief and does not analyze the challenges of using a TEE together with an HSM as in SECV. In follow-up work [74], they develop lightweight ECU attestation mechanisms based on HTAs, focusing on TPMs [3], HSMs, and DICE [19], but not on TEEs.

TEEs in Automotive and Network Solutions. TEEs have been widely used in network security. TrustedGateway [84] and TEEfilter [83] use TrustZone to implement secure Ethernet gateways but do not address the specific constraints of automotive systems. Ethernet adoption for EVN–IVN communication remains slow due to real-time requirements, although it is increasingly used for non-critical traffic and may eventually complement CAN. In our experience, splitting drivers and tracking application context for CAN, SPI, or UART is substantially more complex than for Ethernet; consequently, for Ethernet we later propose a different design based on kernel-bypass techniques. TeeCheck [66], in contrast, mediates all CAN traffic through the secure world and closely resembles SECV in requiring mutual attestation between ECUs and the gateway. Prior work [74] shows this is feasible using per-task HMACs and rate limiting to mitigate DoS, but TeeCheck still assumes a trustworthy Normal World OS. If the OS is compromised, it can suppress Secure World API calls, creating a critical blind spot.

VIII. Conclusion

This paper addressed security challenges in modern automotive platforms, focusing on EVN-originated attacks on safety-critical IVN ECUs. Drawing on real-world attack cases, we identified a common root cause and proposed SECV, a hardware-rooted mechanism for regulating EVN–IVN communication that provides strong security guarantees under automotive real-time constraints with only moderate control-message overhead. SECV is practical to deploy via secure software updates on existing CAN-based platforms and can be extended to automotive Ethernet with minor software changes, leveraging widely deployed hardware.

IX. Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277326); by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2025; by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No. 2021-0-00528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box); by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Graduate School of Artificial Intelligence Semiconductor(IITP-2025-RS-2023-00256081) grant funded by the Korea government(MSIT); by Inter-University Semiconductor Research Center (ISRC), by Korea Planning & Evaluation Institute of Industrial Technology(KEIT) grant funded by the Korea Government(MOTIE) (No. RS-2024-00406121, Development of an Automotive Security Vulnerability-based Threat Analysis System(R&D)); by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government (MSIT) (No.RS-2025-02215590, Development of AI implementation obfuscation technology to prevent information leakage in On-Device AI). Finally, this work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00438729, Development of Full Lifecycle Privacy-Preserving Techniques using Anonymized Confidential Computing)

References

[1] "Cve-2013-6282: Linux kernel api improper input validation vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2013-6282, 2013.

[2] "Cve-2015-5611: Unspecified vulnerability in uconnect before 15.26.1 in certain fiat chrysler automobiles (fca) from 2013 to 2015," https://nvd.nist.gov/vuln/detail/CVE-2015-5611, 2015.

[3] "Information technology – trusted platform module library, parts 1-4," International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), Tech. Rep. ISO/IEC 11889:2015, 2015, parts: 1. Architecture; 2. Structures; 3. Commands; 4. Supporting Routines. [Online]. Available: https://www.iso.org/standard/66510.html

[4] "Cve-2016-2434: Nvidia video driver privilege escalation vulnerability in android," https://nvd.nist.gov/vuln/detail/CVE-2016-2434, 2016.

[5] "SAE J2284: High-Speed CAN (HSC) for Vehicle Applications at 500 kbps," SAE International, Tech. Rep. J2284/4_201606, 2016, available: https://www.sae.org/standards/content/j2284/4_201606/.

[6] "Cve-2020-12769: Race condition in designware spi controller driver," https://ubuntu.com/security/CVE-2020-12769, 2020, ubuntu Security Notice.

[7] "Cve-2020-28588: Linux kernel /proc/pid/syscall stack info leak," 2021, accessed: 2025-07-25. [Online]. Available: https://threatpost.com/linux-kernel-bug-wider-cyberattacks/165640/

[8] "Cve-2021-23907: Heap overflow in headunit ntg6 in the mbux infotainment system," https://nvd.nist.gov/vuln/detail/CVE-2021-23907, 2021.

[9] "Cve-2021-23908: Type confusion in headunit ntg6 (mbux infotainment system, mercedes-benz)," https://www.cvedetails.com/cve/CVE-2021-23908/, 2021.

[10] "Cve-2021-23909: Improper input validation in hermes 2.1 (mbux infotainment system, mercedes-benz)," https://nvd.nist.gov/vuln/detail/CVE-2021-23909, 2021.

[11] "Cve-2022-0435: Linux kernel tipc stack buffer overflow," 2022, accessed: 2025-07-25. [Online]. Available: https://duo.com/decipher/linux-kernel-stack-overflow-patched

[12] "Cve-2022-32292: Connman gweb heap buffer overflow," https://security-tracker.debian.org/tracker/CVE-2022-32292, 2022.

[13] "Cve-2022-32293: Connman wispr use-after-free," https://ubuntu.com/security/CVE-2022-32293, 2022.

[14] "Cve-2022-46152: Op-tee os improper input validation allows local privilege escalation via cleanup_shm_refs()," https://nvd.nist.gov/vuln/detail/CVE-2022-46152, 2022, accessed: 2025-07-25.

[15] "Cve-2023-0179: Linux kernel netfilter stack-based buffer overflow," 2023, accessed: 2025-07-25. [Online]. Available: https://linuxsecurity.com/news/security-vulnerabilities/a-new-privilege-escalation-vulnerability-in-the-linux-kernel-enables-a-local-attacker-to-execute-malware-on-vulnerable-systems

[16] "Cve-2023-41325: Double free in op-tee shdr_verify_signature leading to potential secure world memory corruption," https://nvd.nist.gov/vuln/detail/CVE-2023-41325, 2023, accessed: 2025-07-25.

[17] "Cve-2021-46959: Use after free in linux kernel spi subsystem," https://feedly.com/cve/CVE-2021-46959, 2024, feedly - CVE Advisory and Patch.

[18] "Cve-2024-26807: Pointer handling error in cadence quadspi driver," https://feedly.com/cve/CVE-2024-26807, 2024, feedly - CVE Advisory.

[19] "Dice attestation architecture," Trusted Computing Group, Tech. Rep. Version 1.1, Revision 0.18, January 2024, pUBLISHED. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-Version-1.1-Revision-18_pub.pdf

[20] "Cve-2022-49844: Linux kernel can frame drop in virtual interfaces," https://nvd.nist.gov/vuln/detail/CVE-2022-49844, 2025, nVD Vulnerability Detail.

[21] "Cve-2025-38262: Concurrency bug in linux uart driver registration," https://security-tracker.debian.org/tracker/CVE-2025-38262, 2025, debian Security Tracker.

[22] I. T. AG, "Infineon's aurix tc4xx: World's first iso/sae 21434 certified automotive mcu family," 2022, https://www.infineon.com/cms/en/about-infineon/press/market-news/2022/INFCSS202211-028.html.

[23] ——, "Infineon and vector advance automotive cybersecurity with new microcontroller platform," 2024, https://www.dennemeyer.com/fileadmin/a/media-library/reports/cybersecurity_in_mobility_2024-05.pdf.

[24] T. H. Aldhyani and H. Alkahtani, "Attacks to automatous vehicles: A deep learning algorithm for cybersecurity," *Sensors*, vol. 22, no. 1, p. 360, 2022.

[25] E. Aliwa, O. Rana, C. Perera, and P. Burnap, "Cyberattacks and countermeasures for in-vehicle networks," *ACM computing surveys (CSUR)*, vol. 54, no. 1, pp. 1–37, 2021.

[26] A. Ambekar, R. Schneider, K. Schmidt, and U. Dannebaum, "Future of automotive embedded hardware trust anchors (aehta)," SAE Technical Paper, Tech. Rep., 2022.

[27] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm." in *NDSS*, vol. 16, 2016, pp. 21–24.

[28] S. Basu, M. Staron, J. Horkoff, M. Almgren, C. Berger, and T. Olovsson, "A survey of recent automotive software security vulnerabilities: Trends and attack vectors," *Chalmers University of Technology Research Portal*, 2023, https://research.chalmers.se/publication/543968/file/543968_Fulltext.pdf.

[29] D. Berard and V. Dehors, "Security of connected vehicles," Presentation at GreHack 2023 https://grehack.fr/2023/talks, 2023.

[30] K. Blog, "Vulnerability in remote control systems of kia vehicle," https://os.kaspersky.com/blog/vulnerability-in-kia-car-remote-control-systems/, 2025.

[31] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[32] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "${$rezone$}$: Disarming ${$trustzone$}$ with ${$tee$}$ privilege reduction," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2261–2279.

[33] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX security symposium (USENIX Security 11)*, 2011.

[34] Y. Cho, D. Kwon, H. Yi, and Y. Paek, "Dynamic virtual address range adjustment for intra-level privilege separation on arm." in *NDSS*, 2017.

[35] A. Chowdhury, G. Karmakar, J. Kamruzzaman, A. Jolfaei, and R. Das, "Attacks on self-driving cars and their countermeasures: A survey," *IEEE Access*, vol. 8, pp. 207 308–207 342, 2020.

[36] R. E. Corporation, "Renesas rh850/p1x-c: Automotive safety microcontrollers with embedded hardware security module (hsm)," Tech. Rep., 2016, https://www.renesas.com/en/key-technologies/security/automotive-security.

[37] ——, "Renesas unveils automotive gateway solution based on new r-car s4 socs and pmics," *IoT Now*, 2021.

[38] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao *et al.*, "Strongbox: A gpu tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.

[39] ——, "Strongbox: A gpu tee on arm endpoints," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 769–783.

[40] F.-S. Developers, "F-stack: High performance network framework based on dpdk and freebsd tcp/ip stack," 2017, available at https://github.com/F-Stack/f-stack.

[41] A. N. Europe, "Vw reaches direct supply deals with chipmakers to avoid shortage," https://www.autonews.com/suppliers/vw-will-buy-important-chips-directly-nxp-infineon-renesass/, 2023.

[42] L. Foundation, "Data plane development kit (dpdk)," 2015. [Online]. Available: http://www.dpdk.org

[43] H. Gong, S. Hong, S. Yang, R. Chang, W. Shen, Z. Yuan, C. Yu, and Y. Zhou, "Harness: Transparent and lightweight protection of vehicle control on untrusted android automotive operating system."

[44] L. Guo and F. X. Lin, "Minimum viable device drivers for arm trustzone," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 300–316.

[45] Y. Guo, Z. Wang, B. Zhong, and Q. Zeng, "Formal modeling and security analysis for intra-level privilege separation," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 88–101.

[46] K. Herald, "Autocrypt accelerates as car cyber threats drive global demand," https://www.koreaherald.com/article/10494114, 2025.

[47] S. Hu, Q. Zhang, A. Weimerskirch, and Z. M. Mao, "Gatekeeper: A gateway-based broadcast authentication protocol for the in-vehicle ethernet," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 494–507.

[48] N. Huq, C. Gibson, and R. Vosseler, "Driving security into connected cars: threat model and recommendations," *Trend Micro*, 2020.

[49] T. Instruments, "Texas instruments f29h85x: Real-time mcus for functional safety and cybersecurity in automotive," Tech. Rep., 2024, https://www.embedded.com/ti-introduces-two-new-series-of-real-time-mcus-for-automotive-and-industrial-applications.

[50] J. Jang and B. B. Kang, "Selmon: reinforcing mobile device security with self-protected trust anchor," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 135–147.

[51] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted execution environments: properties, applications, and challenges," *IEEE Security & Privacy*, vol. 18, no. 2, pp. 56–60, 2020.

[52] S. Jeong, M. Ryu, H. Kang, and H. K. Kim, "Infotainment system matters: Understanding the impact and implications of in-vehicle infotainment system hacking with automotive grade linux," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, 2023, pp. 201–212.

[53] P. Jing, Z. Cai, Y. Cao, L. Yu, Y. Du, W. Zhang, C. Qian, X. Luo, S. Nie, and S. Wu, "Revisiting automotive attack surfaces: a practitioners' perspective," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 2348–2365.

[54] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *2010 IEEE symposium on security and privacy*. IEEE, 2010, pp. 447–462.

[55] N. Kühnapfel, C. Werling, H. N. Jacob, and J.-P. Seifert, "Three glitches to rule one car: Fault injection attacks on a connected ev," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, 2025, pp. 1235–1249.

[56] D. Kwon, H. Yi, Y. Cho, and Y. Paek, "Safe and efficient implementation of a security system on arm using intra-level privilege separation," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–30, 2019.

[57] K. S. Lab, "Experimental security assessment of bmw cars: A summary report," Keen Security Lab, Tencent, Tech. Rep., 2018, full technical paper released in 2019. [Online]. Available: https://keenlab.tencent.com/en/whitepapers/Experimental-Security-Assessment-of-BMW-Cars-by-KeenLab.pdf

[58] ——, "Mercedes-benz mbux security research report," Tencent Keen Security Lab, Tech. Rep., 2021. [Online]. Available: https://keenlab.tencent.com/en/whitepapers/Mercedes_Benz_Security_Research_Report_Final.pdf

[59] H. Lee, S. H. Jeong, and H. K. Kim, "Otids: A novel intrusion detection system for in-vehicle network by using remote frame," in *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2017, pp. 57–5709.

[60] S. Lee, S. Kim, C. Song, B. Woo, E. Ahn, J. Lee, Y. Jang, J. Jang, H. Lee, and B. B. Kang, "Genesis: A generalizable, efficient, and secure intra-kernel privilege separation," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, 2024, pp. 1366–1375.

[61] Linux Foundation, "Automotive grade linux (agl)," https://www.automotivelinux.org, 2024, open-source project uniting automakers, suppliers, and technology companies to accelerate automotive software development.

[62] Y. Liu, Z. Yao, M. Chen, A. Amiri Sani, S. Agarwal, and G. Tsudik, "Provcam: A camera module with self-contained tcb for producing verifiable videos," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 588–602.

[63] L. McVoy and C. Staelin, "Lmbench benchmark suite, version 3.0-a9+debian.1-9," https://sourceforge.net/projects/lmbench/, 2025, latest release as of May 2025.

[64] C. Miller, "Lessons learned from hacking a car," *IEEE Design & Test*, vol. 36, no. 6, pp. 7–9, 2019.

[65] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *black hat USA*, vol. 2014, p. 94, 2014.

[66] T. Mishra, T. Chantem, and R. Gerdes, "Teecheck: Securing intra-vehicular communication using trusted execution," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, 2020, pp. 128–138.

[67] C. News, "Carmakers unite to defend against auto hacking," https://www.cbsnews.com/news/carmakers-unite-to-defend-against-auto-hacking/, 2016.

[68] S. Nie, L. Liu, and Y. Du, "Free-fall: Hacking tesla from wireless to can bus," *Briefing, Black Hat USA*, vol. 25, no. 1, p. 16, 2017.

[69] F. J. Niroumand, P. A. Bonab, and A. Sargolzaei, "Security of connected and autonomous vehicles: A review of attacks and mitigation strategies," *SoutheastCon 2024*, pp. 1197–1204, 2024.

[70] NXP Semiconductors, *S32K3 Auto General-Purpose MCUs*, https://www.nxp.com/products/S32K3, 2024, automotive MCUs featuring Hardware Security Engine (HSE) for cryptographic offload, secure key management, and hardware root of trust.

[71] NXP Semiconductors and Ford Motor Company, "Nxp and ford collaborate to deliver next-generation connected vehicles," 2021.

[72] F. Pascale, E. A. Adinolfi, S. Coppola, and E. Santonicola, "Cybersecurity in automotive: An intrusion detection system in connected vehicles," *Electronics*, vol. 10, no. 15, p. 1765, 2021.

[73] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[74] C. Plappert and A. Fuchs, "Secure and lightweight ecu attestations for resilient over-the-air updates in connected vehicles," in *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023, pp. 283–297.

[75] C. Plappert, A. Fuchs, and R. Heddergott, "Analysis and evaluation of hardware trust anchors in the automotive domain," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–11.

[76] C. Plappert, D. Lorych, M. Eckel, L. Jäger, A. Fuchs, and R. Heddergott, "Evaluating the applicability of hardware trust anchors for automotive applications," *Computers & Security*, vol. 135, p. 103514, 2023.

[77] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "${$ftpm$}$: A ${$software-only$}$ implementation of a ${$tpm$}$ chip," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 841–856.

[78] S. Rajapaksha, G. Madzudzo, H. Kalutarage, A. Petrovski, and M. O. Al-Kadri, "Can-mirgu: A comprehensive can bus attack dataset from moving vehicles for intrusion detection system evaluation," in *Symposium on Vehicles Security and Privacy. Internet Society*, 2024.

[79] ReportLinker, "Global and china automotive gateway industry report, 2021-2022," 2022.

[80] A. R. B. ReportLinker Staff, "Navigating cyber risks in auto manufacturing: A 2024 outlook," 2024.

[81] P. M. Research, "Multi-core automotive gateway chip market analysis and forecast 2019-2031," https://pmarketresearch.com/auto/multi-core-automotive-gateway-chip-market/, 2024.

[82] T. Rheinland, "Iso/sae 21434 certification," https://www.tuv.com/world/en/iso-sae-21434-audit-and-certification.html, 2024.

[83] J. Röckl, N. Bernsdorf, and T. Müller, "Teefilter: High-assurance network filtering engine for high-end iot and edge devices based on tees," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1568–1583.

[84] F. Schwarz, "Trustedgateway: Tee-assisted routing and firewall enforcement using arm trustzone," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 56–71.

[85] ScienceAlert, "Remote hijacking fears prompt recall of 1.4 million hackable vehicles," https://www.sciencealert.com/remote-hijacking-fears-prompt-fiat-chrysler-to-recall-1-4-million-hackable-vehicles, 2015.

[86] U. Security, "Upstream's 2025 global automotive cybersecurity report," https://upstream.auto/reports/2025-automotive-smart-mobility-cybersecurity-report-audiobook-edition/, 2025.

[87] ——, "Upstream's 2025 global automotive cybersecurity report," 2025, https://upstream.auto/reports/global-automotive-cybersecurity-report/.

[88] N. Semiconductors, "Nxp s32g3 vehicle network processors: Secure and high-performance solutions for automotive architectures," Tech. Rep., 2024, https://www.nxp.com/products/S32G3.

[89] ——, "S32 automotive processors," https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/s32-automotive-processors:AUTOMOTIVE-MPUS, 2025.

[90] SOCRadar, "Syrus4 iot gateway vulnerability could allow code execution on thousands of vehicles, simultaneously (cve-2023-6248)," https://socradar.io/syrus4-iot-gateway-vulnerability-could-allow-code-execution-on-thousands-of-vehicles-simultaneously-cve-2023-6248/, 2024, a critical vulnerability in the Syrus4 IoT Gateway allows unau-

thenticated remote code execution and control over vehicle ECUs and immobilization.
[91] S. D. I. Software, "Modern automotive cybersecurity through secure communication, strong authentication and flexible firewalls," Tech. Rep., 2024, https://assets.new.siemens.com/siemens/assets/api/uuid:24dfcedc-e2f0-485c-af61-fffe6ec741f0/ca-topic-cybersecurity-en-modern-automotive-cybersecurity-through-secure-communication-white-paper.pdf.
[92] S. Staff, "Bmw patches security flaw that let hackers open doors," https://www.securityweek.com/bmw-patches-security-flaw-let-hackers-open-doors/, 2024.
[93] STMicroelectronics, "Stmicroelectronics stsafe-v and st33-a: Secure automotive hardware for next-gen applications," Tech. Rep., 2024, https://www.st.com/en/secure-mcus/secure-automotive.html.
[94] Synacktiv, "0-click rce on the tesla model3," https://www.synacktiv.com/sites/default/files/2022-10/tesla-hexacon.pdf, 2022.
[95] ——, "Exploiting tesla model 3," https://www.synacktiv.com/sites/default/files/2023-11/tesla-codeblue.pdf, 2023.
[96] ——, "Pwn2own vancouver 2023 tesla exploit chain," https://www.synacktiv.com/sites/default/files/2023-06/SecuriteDesVoitures.pdf, 2023.
[97] ——, "0-click rce on tesla model 3 through tpms sensors," https://www.synacktiv.com/sites/default/files/2024-10/hexacon-0-click-rce-on-tesla-model-3-through-tpms-sensors-light.pdf, 2024.
[98] F. E. S. I. Team, "Kia avoids potential hack of millions of vehicles," https://fieldeffect.com/blog/kia-avoids-potential-hack-of-millions-of-vehicles, 2024.
[99] K. Team. (2025) Vulnerability in remote control systems of kia vehicle. https://os.kaspersky.com/blog/vulnerability-in-kia-car-remote-control-systems/.
[100] T. L. K. Team, "Access to the address space of the process: copy_to_user and related apis," https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html, 2025, accessed: 2025-08-21.
[101] ——, "struct sk_buff - the linux kernel documentation," https://docs.kernel.org/networking/skbuff.html, 2025, accessed: 2025-08-21.
[102] R. Tech, "Top 10 automotive cybersecurity trends 2024," 2024, https://www.rinf.tech/top-10-automotive-cybersecurity-trends-2024/.
[103] VicOne. (2025) Vicone automotive cyberthreat landscape 2025: Preparing for tomorrow's threats. https://www.vicone.com/resources/research-reports/automotive-cyberthreat-landscape-2025.
[104] VicOne and M. Consortium. (2024) Advancing automotive cybersecurity through zero trust architecture. https://vicone.com/blog/advancing-automotive-cybersecurity-through-zero-trust-architecture.
[105] J. Wang, A. Li, H. Li, C. Lu, and N. Zhang, "Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone," in *2022 IEEE symposium on security and privacy (SP)*. IEEE, 2022, pp. 352–369.
[106] B. Weyl, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger *et al.*, "Evita deliverable d3. 2: Secure on-board architecture specification," Technical report, EVITA Consortium, Tech. Rep., 2011.
[107] M. Yan, J. Li, and G. Harpak, "Security research report on mercedes-benz cars," *Black Hat USA*, vol. 38, 2020.
[108] M. Zoppelt and R. Tavakoli Kolagari, "What today's serious cyber attacks on cars tell us: Consequences for automotive security and dependability," 2020, https://opus4.kobv.de/opus4-ohm/frontdoor/deliver/index/docId/1481/file/main.pdf.

APPENDIX A
ARTIFACT APPENDIX

In modern automotive systems, an externally connected platform (EVN) often interfaces with the in-vehicle network (IVN) to perform control functions. Typically, an IVN gateway filters which messages from the EVN are allowed to reach IVN ECUs. However, the IVN gateway lacks visibility into the specific applications running on the EVN and therefore cannot accurately attribute or verify message origins. This allows remote attackers who compromise the EVN platform to masquerade as authorized applications and issue malicious control messages to IVN ECUs—or even to the gateway itself. Our system, SECV, mitigates this threat by introducing a TEE-based security architecture on the EVN platform that mediates all outbound IVN traffic. SECV establishes authentic, application-bound communication paths between EVN applications and the IVN gateway, preventing message spoofing and unauthorized access. At its core, SECV relies on secure message buffers—per-application memory regions allocated by the TEE and used for verified control messages.

A. Description & Requirements

The artifact includes SECV's implementation of the secure message buffers at the EVN platform. We provide our modified Normal world OS (Linux), the Secure World OS (OPTEE), the Secure World Monitor (ARM Trusted Firmware), a sample of the modified FlexCAN driver at the Normal world, and its accompanying Secure World driverlet.

1) How to access: We have open-sourced the artifact to a publicly available GitHub Repository at https://github.com/secv-ndss2026/secv.git, and DOI: https://doi.org/10.5281/zenodo.17785984. We do recommend using the GitHub repository, as it will have more recent updates to the artifact, while the one at the DOI is frozen at the version published at the time of open-sourcing the paper artifact (2025/12).

2) Hardware dependencies: Our current prototype of SECV is built for NXP S32G3. We understand the challenges in acquiring such a board, and are making our best effort to provide a porting for Raspberry Pi 4 as the EVN platform, although this may take some time. However, we believe that by following our implementation for S32G3, it should be effortless to port for Raspberry Pi 4.

3) Software dependencies: To run the artifact on the NXP S32G3 platform, users currently need the NXP HSM firmware, S32 Design Studio (S32DS) IDE, NXP FSL Linux, NXP OP-TEE OS, and NXP ARM Trusted Firmware (ATF). Some of these components require an NXP license. However, for other platforms that already integrate Linux, OP-TEE, and ATF, it should be straightforward to follow our modifications and reproduce the build accordingly.

4) Benchmarks: For performance evaluation, we provide both the microbenchmarks that run in under 2 minutes to test the SECV's performance, especially under stress. These are reported in the paper. For the real-world workload, we use real-world data from the site https://ocslab.hksecurity.net/Datasets, and this is already loaded on the board.

B. Artifact Installation & Configuration

We provide build and execution scripts, along with detailed usage instructions, in the repository. Users can follow these scripts to build the system and reproduce the experiments as described.

C. Experiment Workflow

To evaluate SECV's performance, we provide a comprehensive set of experiments covering both system-level and communication-level performance metrics:

- **E1: Microbenchmarks for Communication Performance** These experiments measure the latency overhead introduced by SECV in message transmission between the EVN and IVN platforms using synthetic workloads. Each microbenchmark runs for approximately one minute to capture stable latency and throughput trends.
- **E2: Real-World Workload Benchmarks** These experiments evaluate SECV's practicality in real automotive communication scenarios by replaying real-world vehicle traffic traces. We measure the impact of SECV on end-to-end latency and throughput of IVN control messages under realistic operating conditions.
- **E3: System Performance Benchmarks** To assess system-level effects, we use LMBench to evaluate the performance impact of SECV's kernel-level modifications, focusing on process management, memory operations, and inter-process communication performance.

We clearly describe in the repository how to use each script to run the experiments as outlined above.

D. Major Claims

SECV enables secure end-to-end communication between EVN platforms while incurring about 6.5% geomean throughput overhead, and latency overhead 3.9%, and demonstrating near-zero message loss even under stress testing (with transmission gaps of $0ms$).

- (C1): SECV incurs minimal communication overhead, about 6.5% geometric mean throughput and latency overhead 3.9%, and exhibits near-zero message loss.
- (C2): Under real-world workloads, SECV ensures that all control messages are transmitted within their required time windows, without any message loss.
- (C3): SECV introduces a modest system performance overhead of 11% (geometric mean), as measured using the *LMBench* benchmark suite.

E. Evaluation

As described above, we provide three experiments to support the claims presented in the paper. The repository includes all necessary scripts along with detailed instructions on how each can be executed to reproduce the reported results. For real-world communication workloads, individual experiments may take between 10 minutes and 2 hours to complete; however, we also provide scaled-down versions that finish within 20 minutes per experiment. Additionally, since SECV demonstrates no message loss for transmission gaps greater than 1 ms, experiments involving significantly larger transmission gaps (e.g., exceeding $10ms$) may be safely skipped. More detailed experimental results are summarized in Tables IV, V, VI, and VII.

Follows an exemplary structure for one experiment (Ey):

1) Experiment (E1): [Microbenchmark for Communication Overhead] [20 human-minutes]: This experiment measures the round-trip communication overhead of control messages exchanged between the EVN platform and the IVN gateway. The EVN platform transmits control messages with CAN ID 0, and for each successfully received message, the IVN gateway responds with a control message using CAN ID 4. Latency is defined as the elapsed time between the transmission of a control message and the reception of the corresponding response message. Message loss is calculated as the difference between the number of transmitted and received control messages, while throughput is measured as the number of control messages transmitted per second. The experiment is repeated with varying control message sizes (8, 16, 32, and 64 bytes) to compute the geometric mean overhead in message throughput and message loss across different message sizes.

2) Experiment (E2): [Communication Overhead over Real-World Workloads] [100–120 human-minutes]: This experiment extends the previous setup by replaying CAN messages recorded from an actual vehicle driven for over five hours. To align with our IVN gateway configuration—which accepts only specific CAN IDs and does not respond to mismatched ones—we modify the recorded message IDs accordingly. To reduce experimental time, the control messages are grouped based on their transmission gaps, and each group is evaluated separately.

3) Experiment (E3): [System Performance Benchmarks] [20–60 human-minutes]: This experiment evaluates the general system performance overhead introduced by SECV using the *LMBench* benchmark suite. Since SECV hooks system calls and partitions the OS into two components, some performance degradation is expected. We measure the overhead of the most affected system calls and functions, including memory mapping and I/O operations. The entire experiment can be executed using a single terminal command, as described in the repository.

[How to: Preparation and Execution] The repository provides scripts for preparing and executing the benchmarks, along with detailed usage instructions.

[Results]

```
Generating report...
############################################################
Tx frames:                49173
Rx frames:                49173
Tx data transfer:         3147072 bytes
Rx data transfer:         3147072 bytes
Tx frames/s:              4917
Rx frames/s:              4917
Tx throughput:            2517 Kbit/s
Rx throughput:            2517 Kbit/s
Lost frames:              0
Lost frames (%):          0.0%
M7_0 core load:           62.6853%
M7_1 core load:           2.20412%
############################################################
```

Fig. 7: Expected results from executing the microbenchmarks for performance measurement include the number of transmitted (Tx) and received (Rx) frames. The results also report throughput—expressed in frames per second (frames/s) or kilobits per second (KBit/s)—and the percentage of message loss.

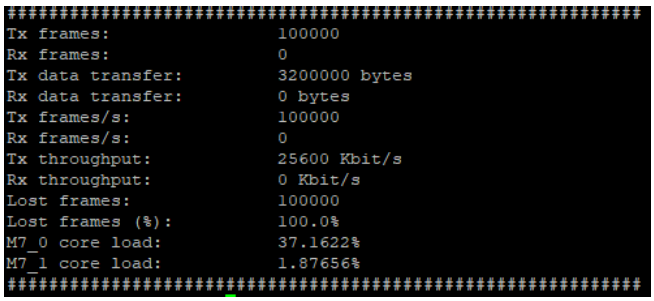


Fig. 8: Expected results from executing the real-world workload experiments for performance measurement include the number of transmitted (Tx) and received (Rx) frames.

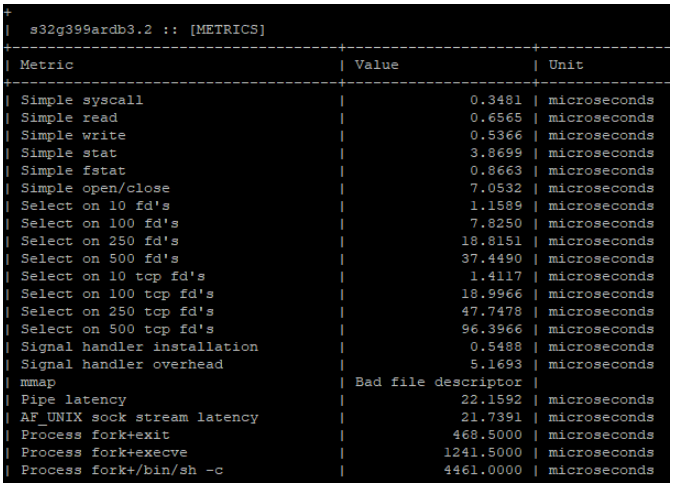


Fig. 9: Expected results from executing the *LMBench* suite include the execution time for each operation, which can be compared against the baseline to compute the normalized overhead.

The interpretation of E2 results follows the same approach as in the microbenchmark experiments. However, depending on the IVN GW setup, if the IVN GW rejects the packets, as shown in the figure above, they will not be echoed back; hence, the results will be similar to those shown in the figure. What matters here is that all the frames are transmitted successfully, and no transmissions miss their deadlines. Check that all the frames in the file being replayed have been transmitted successfully.

Metric	SECV 1	SECV 2	SECV 3	SECV 4	SECV 5	SECV (avg)	SECV (std)
fork+execve	1.574 25	1.558 50	1.514 50	1.517 75	1.524 75	1.537 95	0.03
fork+exit	0.611 67	0.629 25	0.608 22	0.607 10	0.609 11	0.613 07	0.01
write	0.000 54	0.000 54	0.000 54	0.000 54	0.000 54	0.000 54	0.00
open/close	0.007 10	0.007 13	0.007 08	0.007 10	0.007 09	0.007 10	0.00
stat	0.003 76	0.003 76	0.003 73	0.003 75	0.003 78	0.003 76	0.00
fstat	0.000 88	0.000 88	0.000 88	0.000 88	0.000 88	0.000 88	0.00
send	0.003 93	0.004 87	0.003 89	0.004 06	0.004 12	0.004 17	0.00
recv	0.010 64	0.010 79	0.010 64	0.010 88	0.010 88	0.010 77	0.00
mmap	0.064 94	0.055 15	0.056 91	0.063 24	0.064 39	0.060 93	0.00

TABLE IV: LMbench microbenchmark results under SECV. Values are averages over five runs.

Metric	Baseline 1	Baseline 2	Baseline (avg)	Baseline (std)	Overhead (avg)
fork+execve	1.243 80	1.241 50	1.242 65	0.001 63	1.24
fork+exit	0.477 73	0.468 50	0.473 11	0.006 52	1.30
write	0.000 54	0.000 54	0.000 54	0.000 00	1.00
open/close	0.007 01	0.007 05	0.007 03	0.000 03	1.01
stat	0.003 91	0.003 87	0.003 89	0.000 03	0.97
fstat	0.000 87	0.000 87	0.000 87	0.000 00	1.01
send	0.003 07	0.003 05	0.003 06	0.000 17	1.36
recv	0.010 70	0.011 00	0.010 85	0.000 21	0.99
mmap	0.047 82	0.054 18	0.051 00	0.004 50	1.19

TABLE V: Baseline performance and relative SECV overhead (ratio).

Gap (SECV)	Frame Size	Tx (avg)	Rx (avg)	Latency (avg)	Lost frames (%)
0	8	13 985.1	13 987.7	71.5700	11.00
0	16	11 095.2	11 095.2	90.2730	0.00
0	32	7768.6	7768.6	129.0030	0.00
0	64	4862.3	4862.3	206.4070	0.00
1	8	929.0	929.0	1076.0220	0.00
1	16	928.1	928.1	1076.4170	0.00
1	32	928.0	928.0	1076.9150	0.00
1	64	927.0	927.0	1078.0000	0.00

TABLE VI: Communication measurements over 10 runs (SECV).

Gap (Baseline)	Frame Size	Tx (avg)	Rx (avg)	Latency (avg)	Lost frames (%)
0	8	15544.2	14071.9	71.131	9.5
0	16	12059.4	12059.4	83.045	0
0	32	8128.4	8128.4	123.31	0
0	64	4941.5	4941.5	203.046	0
1	8	929	929	1076.022	0
1	16	928.1	928.1	1076.417	0
1	32	928	928	1076.915	0
1	64	927	927	1078	0

TABLE VII: Communication over 10 runs (Baseline).