

USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Short: Race to Charge: Concurrency Bugs and State Integrity Violations in EV Charging Infrastructure

Jaeyeong Lee, *Independent Researcher;* Seonhyeong Lee, *SK Shieldus*

https://www.usenix.org/conference/vehiclesec26/presentation/lee

This paper is included in the Proceedings of the
4th USENIX Symposium on Vehicle Security and Privacy.

August 10–11, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-56-4

Open access to the Proceedings of the
4th USENIX Symposium on Vehicle Security and Privacy
is sponsored by USENIX.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Short: Race to Charge: Concurrency Bugs and State Integrity Violations in EV Charging Infrastructure

Jaeyeong Lee
Independent Researcher

Seonhyeong Lee
SK Shieldus

Abstract

Electric vehicle (EV) charging infrastructure must juggle hardware control, communication protocols, and cloud-based management all at once, relying on complex event-driven multithreaded software underneath. This paper examines concurrency vulnerabilities in `everest-core`, an open-source charging framework hosted under the Linux Foundation Energy umbrella. Through manual code auditing of its threading and callback architecture, combined with sanitizer-assisted reproduction experiments, we uncovered **11** concurrency vulnerabilities, all of which now have public CVE IDs. The vulnerabilities fall into three recurring patterns: inconsistent locking discipline, lack of atomicity in state transitions, and missing state validation on asynchronous events. We confirmed eight data races using TSan, ASan, and UBSan, and the remaining three state integrity violations through logic-based proof-of-concept exploits. These bugs can lead to unauthorized charging that persists after a remote stop command, authorization withdrawal bypass, and denial of service through memory corruption.

1 Introduction

EV charging stations are spreading fast across the globe. According to IEA figures, more than 1.3 million public charging points were added in 2024, bringing the global stock to over 5 million [8], and the software powering them has grown correspondingly complex.

An EVSE (EV Supply Equipment) unit handles CP signaling with the vehicle (IEC 61851, ISO 15118 [9]), cloud communication over OCPP [15], authorization workflows, and power management—all within a single multithreaded stack. Asynchronous events and callbacks coordinate the whole thing, with extensive cross-thread interaction throughout.

Research into EV charging security has branched out in many directions: protocol-level analysis [2, 3], authentication bypass in management systems [14], physical-layer attacks [19], V2G relay attacks [5], and broad threat surveys [1, 10]. Automated approaches like OCPPStorm [6] have also emerged, using protocol message fuzzing to hunt for flaws. Most of this work, however, targets protocol *specifications* or *input handling*. Empirical studies that dig into *implementation-level* concurrency bugs arising from multithreaded execution inside EVSE software remain scarce.

This gap matters because it means implementation-level risks may be underestimated. Concurrency bugs can be triggered even when the protocol itself is correctly implemented—all it takes is legitimate messages arriving at adversarial timing. The consequences range from session state corruption and authorization bypass to memory safety violations, none of which are easily caught by conventional fuzzing or protocol analysis alone.

We focus on `everest-core` [12], an open-source EV charging framework hosted by Linux Foundation Energy. `everest-core` sees real-world deployment in commercial chargers including the Pionix BelayBox, and a stack buffer overflow in its V2GTP implementation was previously reported as CVE-2024-37310 [16]. That said, a systematic empirical examination of concurrency issues across the multithreaded architecture has, to our knowledge, only been reported in limited form.

Our analysis reveals that three recurring concurrency patterns produce exploitable vulnerabilities throughout the modular architecture of `everest-core`: (1) *Inconsistent locking discipline*, where only some code paths acquire a lock before accessing a shared resource; (2) *Lack of state transition atomicity*, where a state change and its associated flags are not updated as a single unit, exposing intermediate states; (3) *Missing state validation*, where incoming asynchronous events get processed without checking whether the current state actually permits them. From these three patterns, we identified **11** concurrency vulnerabilities. All **11** now have public CVE IDs: CVE-2026-33015, CVE-2026-33014, CVE-2026-29044, CVE-2026-26074, CVE-2026-26073, CVE-2026-26072, CVE-2026-26071, CVE-2026-26070, CVE-2026-33009, CVE-2026-27814, and CVE-2026-27813. We confirmed eight data races with TSan [17]/ASan/UBSan and the three state integrity violations with logic-based PoCs.

Contributions.

- We carry out a concurrency vulnerability analysis of an open-source EV charging stack, identifying **11** vulnerabilities with **11** public CVE IDs, and verify their impact through reproduction experiments. Unlike prior work that concentrates on protocol specifications and input handling, our study targets timing-dependent flaws in the multithreaded runtime.
- We show that the discovered vulnerabilities stem from three recurring patterns and trace these back to structural characteristics of the event-driven modular architecture.
- We confirm every finding through sanitizer verification or logic PoCs, and present attack scenarios encompassing unauthorized charging continuation, authorization bypass, and denial of service.

2 Background and Threat Model

2.1 EV Charging Architecture

An EVSE communicates with two external entities. One is the *electric vehicle* (EV) via CP signaling (IEC 61851) or ISO 15118; the other is the *Charging Station Management System* (CSMS) over an OCPP WebSocket connection. Internally, four subsystems cooperate: hardware control (*BSP*), the charging state machine (*EVSE Manager*), authorization (*Auth Handler*), and cloud communication (*OCPP module*).

The `everest-core` framework (release 2025.12.1) realizes this architecture as a modular system where components talk to each other through typed interfaces and callbacks. Figure 1 illustrates the concurrent execution architecture along with the relevant attack paths.

A charging session's lifecycle is governed by the state machine inside the `Charger` class. Transitions from `Idle` through `Charging` to `Finished` run on a dedicated thread, but OCPP callbacks, MQTT handlers, authorization responses, and EV information updates each arrive on their own separate threads.

2.2 Threat Model

We consider attackers who can influence one or more legitimate EVSE input channels, but who do not have code execution inside the EVSE process. Table 1 aligns each finding with the attacker-visible channel needed to trigger it. The three main channels are: a compromised or malicious CSMS that can send OCPP commands; an EV-side adversary that can shape CP/ISO 15118 timing, plug/unplug behavior, or BCB toggles; and an exposed or tunneled MQTT/API management plane. These avenues reflect realistic deployment threats: CSMS account compromise [3] and on-site LAN access at public charging stations have both been documented.

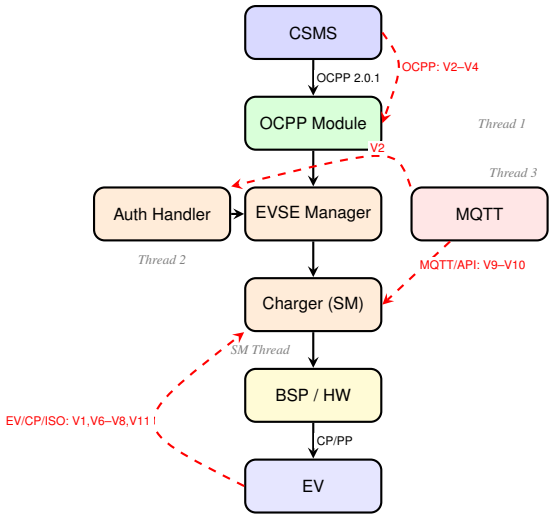


Figure 1: Concurrent execution architecture of `everest-core`. Solid arrows indicate normal data flow; red dashed arrows mark attacker-facing input channels that can shape event timing. The resulting races occur inside callback and state-machine threads.

Table 1: Attacker-facing channels required for each finding.

ID	Channel	Attacker requirement
V1	CSMS + EV/CP	RemoteStop plus EV-side BCB timing during `StoppingCharging`.
V2	CSMS + MQTT/API	OCPP stop request plus delayed Auth response via management plane; not CSMS-only.
V3	CSMS/API timing	WithdrawAuthorization before TransactionStarted is reflected in AuthHandler.
V4	CSMS/OCPP	Firmware or log status events racing with startup event buffering.
V5	Startup event	OCPP 1.6 power-meter key event during initialization.
V6,V8	EV info + meter	SoC update, meter callback, and session-end timing; can occur during normal charging.
V7	EV/ISO info	EVCCID update racing with session processing.
V9,V10	MQTT/API	Phase-switch command reaching Charger while the state-machine thread runs.
V11	EV/Auth timing	Plug-in, session, and authorization timeout paths racing on the Auth timer.

The attacker's goal is to exploit concurrency bugs through *adversarial timing* of otherwise legitimate interactions. We explicitly distinguish externally triggerable bugs from races that primarily represent reliability hazards under normal operation. In particular, V2 combines an OCPP RemoteStop with a delayed authorization response on `everest-core`'s internal Auth–EVSE command plane; we treat it as remotely triggerable when the MQTT/API plane is reachable, but not as a pure CSMS-only exploit.

3 Methodology

We conducted the analysis against `everest-core` release 2025.12.1 in three phases.

Phase 1: Identifying the Concurrency Attack Surface.

Table 2: Thread-to-shared-resource mapping. R = read, W = write, P = mutex-protected, U = unprotected. The OCPP column encompasses both OCPP callbacks and EV info / power meter callbacks.

Shared Resource	SM	OCPP	MQTT	Auth
`shared_context`	R/W P	–	W U	–
`transaction_active`	R/W P	R	–	R
`authorized`	R/W P	–	–	W U
`event_queue`	–	R/W P/U	–	–
`evse_soc_map`	–	R/W U	–	–
`evse_evcc_id`	–	R/W U	–	–
`timeout_timer`	–	–	–	W U

We analyzed four core module groups—EVSE Manager, OCPP 1.6, OCPP 2.0.1, and Auth Handler—and identified five distinct execution contexts: (1) the state machine thread (`Charger::main_thread`), (2) OCPP callback threads (WebSocket receiver), (3) the MQTT subscription handler thread, (4) authorization response threads, and (5) EV info / power meter callback threads. Context (5) fires inside the OCPP module when power meter or EV information updates come in. It shares a process with the OCPP callback thread (2) but forms a separate callback chain, making it an independent source of races. We mapped seven shared state fields accessed across these contexts; Table 2 summarizes the key thread-to-resource mappings. The OCPP column in the table covers both (2) and (5), reflecting the fact that these two contexts access the same shared resources but compete through different callback paths.

Phase 2: Systematic Code Review. For each identified shared resource, we traced every access path and checked whether it was lock-protected. Our review concentrated on three bug patterns: (1) *Inconsistent locking*: some paths acquire a mutex for a given field while others skip it entirely; (2) *Lack of state transition atomicity*: a state change and the flags it depends on get updated at separate points, leaving an observable intermediate state; (3) *Missing state validation*: an incoming event gets processed without verifying that the current state permits it.

Bug Determination Criteria. We classified a suspect site as a vulnerability when: (i) concurrent accesses to the same field exist from different execution contexts, (ii) at least one access is a write, and (iii) no shared lock serializes the two accesses, or the locking is applied inconsistently. For state integrity violations, criterion (ii) was replaced with "there exists an event sequence that permits a semantically invalid transition from the current state."

Phase 3: PoC Construction and Verification. We wrote Google Test–based PoCs for every suspect bug. For concurrency memory defects (V4–V11), we spawned concurrent threads to induce the race condition and verified the outcome with TSan (happens-before violations), ASan (heap-use-after-free), and UBSan (loading of invalid values). For state integrity violations (V1–V3), we replayed specific event sequences and asserted that the state machine reached an invalid state—these are logic-based PoCs. For V2, we additionally reproduced the sequence on real ARM hardware using a Raspberry Pi 4 deployment. This is not an arbitrary test board choice: `everest-core` is publicly packaged for Raspberry Pi via the `everest-demo` snap [18]. On the Pi, we ran the unmodified `everest-core` 2025.12.1 manager together with SIL EVSE modules; the CSMS side issued `RequestStopTransaction`, while the delayed authorization response was injected through the `everest-core` MQTT/API command plane. Across three runs with injected delays of 30 ms, 50 ms, and 70 ms, the stop request was accepted in all cases, yet no OCPP `TransactionEvent(Ended)` and no local `Transaction Finished` transition were observed. This experiment validates the state-machine failure in a hardware-backed deployment of the stack, but it should not be interpreted as a CSMS-only trigger.

4 Findings

We found **11** concurrency vulnerabilities spanning three categories. Table 3 provides a full summary. The corresponding public CVE IDs are CVE-2026-33015, CVE-2026-33014, CVE-2026-29044, CVE-2026-26074, CVE-2026-26073, CVE-2026-26072, CVE-2026-26071, CVE-2026-26070, CVE-2026-33009, CVE-2026-27814, and CVE-2026-27813. The "Access Req." column denotes the attacker-visible input channel or trigger precondition required to reach the vulnerable interleaving, thereby aligning each finding with the threat model in Section 2.2. Entries may refer either to an attacker-facing channel (e.g., CSMS, MQTT/API, EV/CP/ISO) or to the minimal triggering precondition when no dedicated external channel is required (e.g., normal operation or startup timing).

4.1 State Integrity Violations

V1 through V3 are logic-level flaws where the charging session state machine permits semantically invalid transitions under specific timing. We describe V2 in detail as a representative case.

V2: Delayed Authorization Response Defeats RemoteStop. The function `cancel_transaction()` sets `authorized = false` as part of the stop sequence. But if a delayed response to a previously issued authorization request arrives after the RemoteStop has been processed, `authorize(true)` restores `authorized = true` without checking the current state. When the EV subsequently disconnects or the contactor-open event is processed, the code may see that `authorized` is true and skip the `stop_transaction()` call, leaving the transaction hanging open. Figure 2 depicts this event sequence. The root cause maps to the third pattern (*missing state validation*): `authorize()` unconditionally sets the

Table 3: Summary of concurrency vulnerabilities discovered in `everest-core`. **Category**: A = state integrity violation, B = shared container data race, C = external-input cross-thread race. **Confirmed**: T = TSan, A = ASan, U = UBSan, L = logic PoC. **Root Cause**: P1 = inconsistent locking, P2 = lack of atomicity, P3 = missing state validation. **Access Req.** maps each finding to the corresponding attacker-facing channel or trigger precondition. V6–V8 can also occur under certain timing races during normal operation. The table intentionally excludes independently reported non-concurrency input-validation bugs.

ID	Component	Trigger Summary	Cat.	Root Cause	Access Req.	Impact	Conf.	CVE
V1	Charger	BCB toggle after RemoteStop	A	P2	CSMS+EV timing	Unauth. charge cont.	L	CVE-2026-33015
V2	Charger	Delayed AuthResp after RemoteStop	A	P3	CSMS+MQTT/API	Billing integrity viol.	L	CVE-2026-33014
V3	AuthHandler	WithdrawAuth before TransactionStarted	A	P3	CSMS/API timing	Auth. bypass	L	CVE-2026-29044
V4	OCPP 2.0.1	FW/log callback pushes without lock	B	P1	CSMS network	Crash	T	CVE-2026-26074
V5	OCPP 1.6	Power meter key callback pushes w/o lock	B	P1	Startup event	Crash	A,U	CVE-2026-26073
V6	OCPP 1.6	Concurrent R/W/reset on SoC map	B	P1	Normal operation	Exception	A	CVE-2026-26072
V7	OCPP 2.0.1	Concurrent R/W on EVCC ID string	B	P1	Normal operation	Memory corruption	A	CVE-2026-26071
V8	OCPP 2.0.1	Concurrent R/W/reset on SoC map	B	P1	Normal operation	Exception	T	CVE-2026-26070
V9	Charger	MQTT phase-switch writes without lock	C	P1	MQTT (LAN)	State corruption	T,U	CVE-2026-33009
V10	Charger	Additional field accesses in same method	C	P1	MQTT (LAN)	UB	T,A,U	CVE-2026-27814
V11	AuthHandler	Concurrent timer stop/timeout invocation	C	P1	EV/Auth timing	Crash	A	CVE-2026-27813

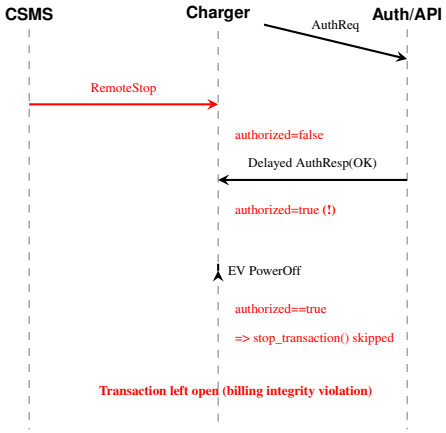


Figure 2: V2 attack sequence. A delayed authorization response, delivered through the internal Auth/MQTT command plane in our deployment experiment, reactivates the `authorized` flag so `stop_transaction()` may not fire when the EV disconnects.

flag even in the `StoppingCharging` state. We further observed the same behavior on a Raspberry Pi 4 deployment of `everest-core` 2025.12.1: even though the charge point returned `Accepted` to `RequestStopTransaction`, the transaction remained open after the delayed authorization response was injected. V1 is a case where `cancel_transaction()` does not clear `transaction_active` right away (*lack of atomicity*), allowing a BCB toggle to restart the session. V3 involves a timing window in asynchronous event delivery where a WithdrawAuthorization takes the deauthorize-only path instead of the stop-transaction path.

4.2 Shared Container Data Races

V4 through V8 involve unprotected concurrent access to `std::map`, `std::queue`, `std::optional`, and `std::string` fields from multiple callback threads.

Listing 1: V4: Locking inconsistency on the same event queue. The error handler acquires the mutex; the firmware callback does not.

```
// Error handler - correctly locked
void handle_session_event(...) {
  std::scoped_lock lock(session_event_mutex);
  event_queue[id].push(event); // Protected
}
// Firmware callback - no lock!
void handle_firmware_status(...) {
  // No mutex acquired!
  event_queue[id].push(fw_event); // Unprotected
}
```

V4: Event Queue Locking Inconsistency. The OCPP 2.0.1 module uses a queue of type `std::map<int32_t, std::queue<variant>>` to buffer events before the stack has finished initializing (`started == false`). The error event handler dutifully acquires `session_event_mutex`, but the firmware update status and log status callbacks push to the same queue without grabbing any lock at all. Listing 1 shows the inconsistency side by side. The race window is especially wide during module initialization: callbacks start arriving while the OCPP stack is still setting up its WebSocket connection, and every path buffers into the same queue yet locking is applied unevenly. TSan flagged concurrent read/write operations on the internal block pointers of `std::deque`.

V5 reproduces the same pattern in the OCPP 1.6 module, this time in the power meter public key callback; ASan/UBSan caught misaligned address access errors. V6–V8 involve completely unprotected accesses to `evse_soc_map` and `evse_evcc_id`, where a power meter callback (reading), an EV info callback (writing), and a session-end handler (resetting) all race against each other. In V7, the race on a `std::string` triggers heap-use-after-free (ASan confirmed). In V6 and V8, the optional gets invalidated between a `has_value()` check and the subsequent `value()` call, throwing `std::bad_optional_access`.

4.3 External-Input Cross-Thread Races

V9 through V11 are triggered by external inputs that reach shared state without first acquiring the state machine mutex.

V9: MQTT Phase-Switch Race. The MQTT message `switch_three_phases_while_charging` calls a method that reads and writes several fields of `shared_context` without holding `state_machine_mutex`. Meanwhile, the state machine loop accesses the exact same fields while holding the mutex. TSan reported four data races, and UBSan caught the loading of an invalid value (190) for a `bool` field. Under the C++ standard, only 0 and 1 are valid representations for `bool`. A value of 190 is one instance of an anomalous value that may be associated with unsynchronized concurrent writes. Such anomalous values can lead to unexpected control flow or outright crashes. Because the phase-switch logic determines the power delivery mode (single-phase vs. three-phase AC), corrupted state values here can propagate into erratic behavior in the higher-level control logic.

V11 is a heap-use-after-free in the Boost.Asio–based timer of `AuthHandler`. The `timeout()` and `stop()` functions get called concurrently from multiple code paths—plug-in events, authorization responses, session termination, deauthorization—and the timer callback ends up referencing a destroyed `io_context` (ASan confirmed).

5 Security Impact

The security consequences of the discovered vulnerabilities fall into three types.

Unauthorized Charging Continuation and Billing Integrity Violation. V1 lets a session restart through a BCB toggle during the `StoppingCharging` window after a RemoteStop, keeping the charger energized. V2 is different in mechanism but similar in outcome: a delayed authorization response derails the stop sequence, leaving the transaction unclosed. In our deployment experiment, the stop command was sent over OCPP and the delayed authorization response was delivered over the `everest-core` MQTT/API command plane. Both undermine the integrity of billing records.

Authorization Bypass. With V3, a session can persist even after a WithdrawAuthorization command. Picture an operator revoking a stolen RFID card's authorization—the command appears to succeed, yet charging continues uninterrupted.

Denial of Service and State Corruption. V4, V5, V7, and V11 can crash the EVSE process via heap corruption or use-after-free. V6–V8 pose reliability risks that can surface under certain timing conditions during normal charging sessions; an attacker who induces rapid plug/unplug cycles can convert them into intentional denial of service. V9–V10 corrupt the internal state of the charging state machine through MQTT message injection, and the corrupted values can propagate into erratic behavior at higher control layers.

We experimentally verified that the scenarios above are reproducible through adversarial timing of legitimate event paths, without code execution on the EVSE process. Some paths are reachable from a CSMS, while others require EV-side timing or access to the `everest-core` MQTT/API command plane. V6–V8 deserve particular note: they can manifest without any external attack under certain timing races, posing an availability risk on their own.

6 Discussion

Architectural Lessons from Recurring Patterns. Three patterns recur across all **11** findings. Inconsistent locking discipline shows up in V4–V11 (8 out of 11), lack of state transition atomicity in V1 and V2, and missing state validation in V2 and V3. These point to a structural vulnerability tendency tied to the framework's design.

Why is inconsistent locking so dominant at 8 out of 11? The answer lies in `everest-core`'s modular structure. Each module manages its own locking policy independently, so when access paths to a shared resource cross module boundaries, some paths end up without a lock. This observation aligns with the atomicity violation and order violation patterns that Lu et al. [13] documented in general-purpose software, and it underscores the need for principled design measures—such as wrapping cross-module shared state behind thread-safe accessors.

The lack of atomicity (V1, V2) and missing state validation (V2, V3) share a common root cause. The state transitions of a charging session are distributed across multiple flags—`authorized`, `transaction_active`, `current_state`—but these flags never get updated as a single atomic unit. The problem was most likely to appear where (i) state transitions and their associated flags are scattered across multiple update sites, and (ii) asynchronous callbacks do not adequately account for what state the system is actually in.

These patterns were observed in `everest-core` and may arise in similar event-driven architectures. We performed a lightweight architectural inspection of other open-source EV charging stacks, including OpenEVSE WiFi, Java-OCA-OCPP, WWCP_OCPP, and SteVe. We did not conduct full vulnerability audits or PoC validation for those projects. However, several of them exhibit the same pattern-level precondition: multiple asynchronous control paths, such as OCPP, HTTP, MQTT, scheduler logic, or WebSocket session handling, converge on shared EVSE, session, or transaction state. This suggests that the architectural conditions behind our findings are not unique to `everest-core`, although we do not claim that the same vulnerabilities are present in those projects.

Our analysis was primarily manual, but parts of the workflow are amenable to automation. Callback entry-point discovery, shared-state inventory construction, lockset comparison, and cross-channel state-update detection can be implemented

as static-analysis heuristics. The harder step is exploitability validation: deciding whether a flagged interleaving violates an EV-specific invariant such as RemoteStop irreversibility, authorization finality, or transaction accounting still requires domain knowledge and targeted replay. A natural direction for future work is therefore a domain-specific concurrency testing framework for EVSE software, combining static candidate generation with schedule fuzzing over OCPP, MQTT/API, and EV/CP/ISO event channels.

Comparing with the concurrency bug characteristics that Li et al. [11] reported for interrupt-driven embedded software, our findings show analogous patterns manifesting at the thread level rather than the interrupt level.

Why EV Charging Software Is Particularly Susceptible. Three structural characteristics of EV charging stacks make concurrency bugs more consequential than in typical networked services. First, state transitions directly govern *real-time hardware control*—contactor switching, CP signaling, and power delivery—so a race does not merely corrupt data but can sustain or interrupt physical energy flow. Second, *multiple independent protocol stacks* (OCPP, ISO 15118, MQTT) each inject events from their own threads, creating a structurally wider cross-thread interaction surface than single-protocol servers. Third, a single state machine binds *authorization, billing, and power delivery* together; any intermediate state exposed by a race therefore has immediate financial consequences (e.g., an unclosed transaction) or safety implications (e.g., unauthorized energization). In contrast, a typical web service experiencing a data race risks database inconsistency, whereas the same class of bug in an EVSE directly affects physical power output and billing integrity.

Limitations. This analysis covers a single open-source framework, `everest-core`. Commercial EVSE implementations may exhibit different concurrency patterns. We validated V2 on a Raspberry Pi 4 hardware deployment with SIL EVSE modules, but we did not test against production EVSE hardware with real contactors, PLC modems, and grid-side power electronics. Given the code-review-based nature of our methodology, additional concurrency bugs may well exist.

7 Related Work

EV charging security research has advanced along several fronts. On the protocol analysis side, Baker and Martinovic [3] studied OCPP message forgery and replay threats, while Alcaraz et al. [2] covered threats and countermeasures for charging infrastructure 4.0. OCPPStorm by Coppoletta et al. [6] applied black-box fuzzing to find message processing defects. All of these focus on message content and format rather than multithreaded execution timing. At the physical layer, Son et al. [19] worked on signal injection defense, Conti et al. [5] demonstrated relay attacks, and Becker et al. [4] conducted V2G threat analysis. Broader threat taxonomies appear in the surveys by Johnson et al. [10], Alashjaee et al. [1], and Nasr et al. [14]. Our work complements these efforts by showing that implementation-level defects rooted in multithreaded execution timing can persist even when the protocol is correctly realized.

In the concurrency analysis space, the atomicity violation and order violation patterns identified by Lu et al. [13] map directly onto our V1–V3 and V4–V10 findings. Wang et al. [20] detected races in interrupt-driven embedded systems using a combined static/dynamic approach, and Li et al. [11] empirically classified concurrency bugs in interrupt-driven software. TSan [17] and RacerX [7] are core techniques we relied on for verification and exploration.

To our knowledge, empirical studies of concurrency vulnerabilities in EV charging stacks have only been reported in limited form.

8 Conclusion

This paper presents a concurrency vulnerability analysis of the `everest-core` EV charging framework, uncovering **11** flaws with **11** public CVE IDs. The vulnerabilities cluster around three patterns—inconsistent locking discipline, lack of state transition atomicity, and missing state validation—and their consequences include unauthorized charging continuation, authorization bypass, and denial of service. The predominance of inconsistent locking at 8 out of 11 cases is particularly telling: it signals that modular architectures lacking a centralized locking policy for shared state are structurally prone to concurrency bugs. This risk factor is likely not unique to `everest-core` and may recur in other event-driven multithreaded systems of similar design.

Artifact Availability. The artifact associated with this paper is permanently archived at https://doi.org/10.5281/zenodo.19548057.

References

[1] Abdullah M. Alashjaee, Abdulhaq Mohammad, and Amer A. Alsulami. A comprehensive survey of cybersecurity threats, attacks, and effective countermeasures in electric vehicle charging infrastructure. *World Electric Vehicle Journal*, 15, 2024.

[2] Cristina Alcaraz, Javier Cumplido, and Alicia Triviño. OCPP in the spotlight: Threats and countermeasures for electric vehicle charging infrastructures 4.0. *International Journal of Information Security*, 22, 2023.

[3] Richard Baker and Ivan Martinovic. OCPP security: Exploring the attack surface of the Open Charge Point Protocol. In *Proceedings of the IEEE Symposium on Security and Privacy Workshops (SPW)*, 2023.

[4] Hanno Becker, Martin Serror, Martin Henze, and Klaus Wehrle. Threat analysis of the vehicle-to-grid charging ecosystem. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS) Workshops*, 2021.

[5] Mauro Conti, Denis Donadel, Radha Poovendran, and Federico Turrin. EVExchange: A relay attack on electric vehicle charging system. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2022.

[6] Gaetano Coppoletta, Rigel Gjomemo, Amanjot Kaur, Nima Valizadeh, V. N. Venkatakrishnan, and Omer Rana. OCPPStorm: A comprehensive fuzzing tool for OCPP implementations. In *Proceedings of the Symposium on Vehicle Security and Privacy (VehicleSec)*, 2024.

[7] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[8] International Energy Agency. Global EV outlook 2025: Electric vehicle charging. https://www.iea.org/reports/global-ev-outlook-2025/electric-vehicle-charging, 2025.

[9] International Organization for Standardization. ISO 15118: Vehicle to grid communication interface, 2022.

[10] Jay Johnson, Timothy Berg, Benjamin Anderson, and Brian Wright. Review of electric vehicle charger cybersecurity vulnerabilities, potential impacts, and defenses. *Energies*, 15, 2022.

[11] Chenyang Li, Rui Chen, Bo Wang, Zhengyu Wang, Ting Yu, Yu Jiang, Bin Gu, and Mengfei Yang. An empirical study on concurrency bugs in interrupt-driven embedded software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[12] Linux Foundation Energy. EVerest: An open source modular framework for EV charging infrastructure. https://github.com/EVerest/everest-core, 2024.

[13] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[14] Tony Nasr, Sadegh Torabi, Elias Bou-Harb, Carol Fung, and Chadi Assi. Power jacking your station: In-depth security analysis and exploitation of electric vehicle charging station management systems. *Computers & Security*, 112, 2023.

[15] Open Charge Alliance. Open Charge Point Protocol (OCPP) 2.0.1 specification. https://openchargealliance.org/, 2022.

[16] PlaxidityX. CVE-2024-37310: EVerest v2gtp implementation stack buffer overflow. https://nvd.nist.gov/vuln/detail/CVE-2024-37310, 2024.

[17] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[18] Snapcraft. Install EVerest - EV charging open-source stack on raspberry pi. https://snapcraft.io/install/everest-demo/raspbian, 2024.

[19] Soyeon Son, Kyungho Joo, Wonsuk Choi, and Dong Hoon Lee. Securing EV charging system against physical-layer signal injection attack. In *Proceedings of the Symposium on Vehicle Security and Privacy (VehicleSec)*, 2024.

[20] Yu Wang, Fengjuan Gao, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. Automatic detection, validation, and repair of race conditions in interrupt-driven embedded software. *IEEE Transactions on Software Engineering*, 48(1), 2022.

USENIX VehicleSec '26 Artifact Appendix: Race to Charge: Concurrency Bugs and State Integrity Violations in EV Charging Infrastructure

Jaeyeong Lee
Independent Researcher
lee@jaeyeong.cc

Seonhyeong Lee
SK Shieldus
leeseon4@gmail.com

A Artifact Appendix

A.1 Abstract

This artifact accompanies "Race to Charge: Concurrency Bugs and State Integrity Violations in EV Charging Infrastructure." It contains the source snapshot used in the paper, a lightweight scanner that codifies the paper's review heuristics for callback/shared-state concurrency bugs, and the Raspberry Pi 4 replay harness used for the hardware-backed validation of V2 / CVE-2026-33014. The artifact is intended for local source-code inspection and local SIL/testbed replay only; it must not be run against production charging stations or infrastructure connected to real vehicles or power electronics.

A.2 Description and Requirements

Artifact location. The frozen artifact is archived at:

https://doi.org/10.5281/zenodo.19548057

The unpacked archive contains `artifact_available/scanners/` (heuristic scanner and README), `artifact_available/poc/` (Raspberry Pi replay for V2 / CVE-2026-33014), `artifact_available/docs/` (setup notes and result summary), and `artifact_appendix/` (this appendix source).

Badge. This submission was evaluated for and awarded **Artifacts Available**. We did not apply for the Functional or Results Reproduced badges.

Hardware. The scanner can be inspected on a normal Linux workstation. The full replay requires a Raspberry Pi 4; our reference platform was a Raspberry Pi 4 Model B (Broadcom BCM2711, 4 ARM Cortex-A72 cores, 4 GB RAM) running arm64 Ubuntu 24.04. No GPU, FPGA, PLC modem, oscilloscope, production EVSE, vehicle, or power electronics are required.

Software. The replay requires Linux, Python 3.10 or newer, Python packages `paho-mqtt`, `PyYAML`, and `websockets`, a local Mosquitto or compatible MQTT broker on `127.0.0.1:1883`, and an in-place `everest-core` 2025.12.1 build with `build/dist/bin/manager`. The scanner itself requires only Python 3 and the checked-out source tree. No external dataset, trained model, or downloaded benchmark is used.

A.3 Set-up

Commands below assume the evaluator starts from the root of the unpacked artifact archive.

1. Install replay dependencies: `cd artifact_available/poc` and `python3 -m pip install -r requirements.txt`.
2. Start a local broker, for example `mosquitto -d -p 1883`.
3. Build `everest-core` if needed: `cmake -S . -B build -GNinja -DCMAKE_BUILD_TYPE=RelWithDebInfo` followed by `cmake --build build -j4`.
4. Run the scanner smoke test: `cd artifact_available/scanners` and `python3 check_concurrency_patterns.py --json /tmp/everest-candidates.json`.

The expected scanner output mentions candidate sites involving `event_queue`, `evse_soc_map`, `evse_evcc_id`, `shared_context.authorized`, `shared_context.transaction_active`, and `timer.callback`, and writes the JSON report to `/tmp/everest-candidates.json`. To check replay dependencies, run `python3 -c "import yaml, websockets"` and `python3 -c "import paho.mqtt.client"`.

A.4 Claims and Evaluation Workflow

This appendix is included for the Available badge; the following workflow summarizes the validation material shipped with the artifact.

C1. The scanner replays the paper's review heuristics on `everest-core` 2025.12.1 and rediscovers the main source-code patterns underlying the confirmed findings.

C2. The V2 / CVE-2026-33014 state-integrity failure can be replayed on a Raspberry Pi 4 deployment of the unmodified `everest-core` manager with SIL EVSE modules.

E1: Scanner replay. Run `check_concurrency_patterns.py` from `artifact_available/scanners/`. The mapping from reported candidate classes to the paper's findings is documented in `artifact_available/scanners/README.md`.

E2: Single V2 replay. After completing setup, run `f2_remote_stop_bypass_poc.py` from `artifact_available/poc/` with a 50 ms delayed authorization response. The successful run prints `RESULT VULNERABLE_REMOTE_STOP_BYPASSED`; the generated logs should show an accepted stop request without a subsequent local `Transaction Finished` or OCPP `TransactionEvent(Ended)`.

E3: Delay sweep. Repeat the replay with the 30/50/70 ms delays used for the paper revision. The expected results are summarized in `artifact_available/docs/results_summary.md`.

A.5 Reusability and Version

The scanner is intentionally heuristic rather than sound or claim-complete, so it is most useful as a regression aid for future `everest-core` versions and nearby EVSE source trees. The replay harness exposes timing and environment parameters such as `--delay-ms`, `--base`, `--run-dir`, and `--port` so that it can be integrated into other local testbeds without source edits.

Based on the LaTeX template for Artifact Evaluation V20260101. Submission, reviewing, and badging methodology for this artifact evaluation can be found at https://secartifacts.github.io/vehiclesec2026/.