Enabling Real-Time Restoration of Compromised ECU Firmware in Connected and Autonomous Vehicles

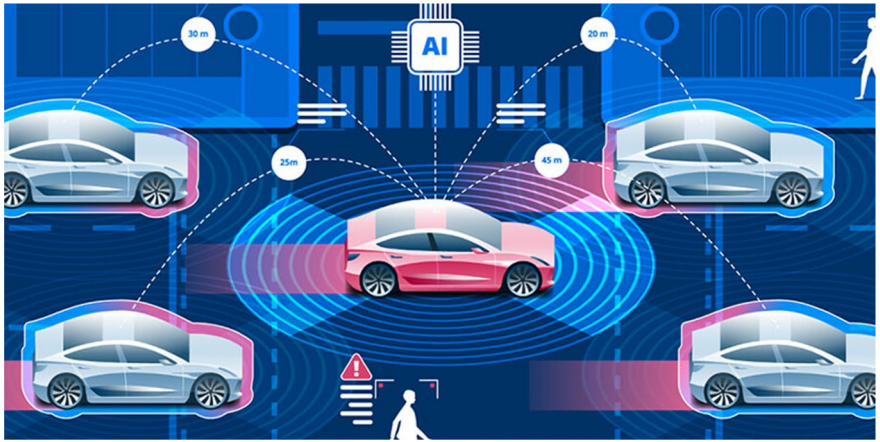
Josh Dafoe, Harsh Singh, **Niusen Chen**, Bo Chen
Department of Computer Science
Michigan Technological University

MICHIGAN TECH
HUSKIES

Motivation

Connected and Autonomous Vehicles (CAVs)

- CAVs will help humans achieve safe, efficient, and autonomous transportation systems
- The risk of cyber threats on CAVs is increasing



CAV is Vulnerable to Several Attacks

- 170 attacks reported on CAVs from 2010 to 2018 (60 in 2018)
  - Remote hack through infotainment
  - CAN access through OBD-II (remote or physical)
  - LiDAR/Radar Spoofing
  - Network denial of service
- Attacks on electronic control units (ECUs)
  - ECUs are widely deployed in CAVs
    - Fuel supply, brake system, ignition, idle speed, etc.
  - Inject malicious code into the internal firmware of ECUs
    - OBD-II
    - Over-the-air (OTA)

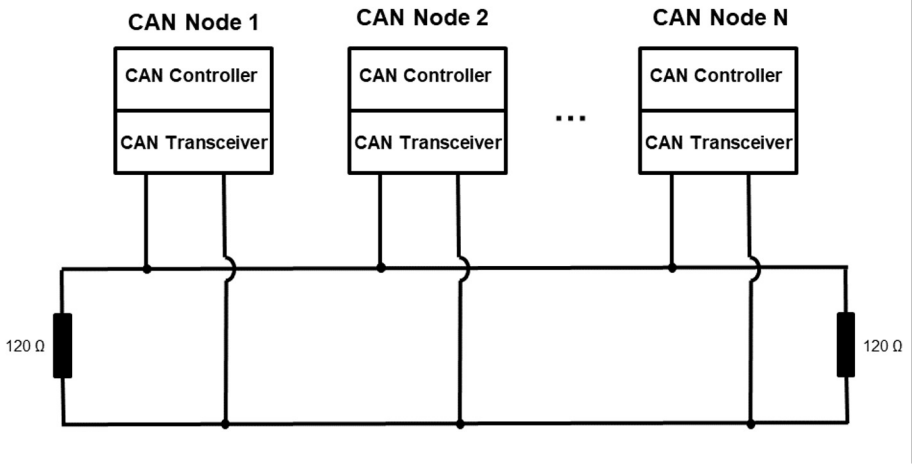
Existing Defenses are not Sufficient

- Current approaches mainly focus on detecting malicious CAN activities
  - The malicious ECU can be detected by monitoring the messages on the CAN bus (via intrusion detection and signal analysis), but no restoration of the firmware is performed
- In this work, we utilize local hardware associated with the steganography to enable ECU firmware restoration in real time
  - Flash memory can guarantee that the old version of firmware is recoverable due to the out-of-place update feature
  - TrustZone helps to defend against the attacker who can compromise the ECU OS
  - Steganography is used to protect the communications among different parts

Background

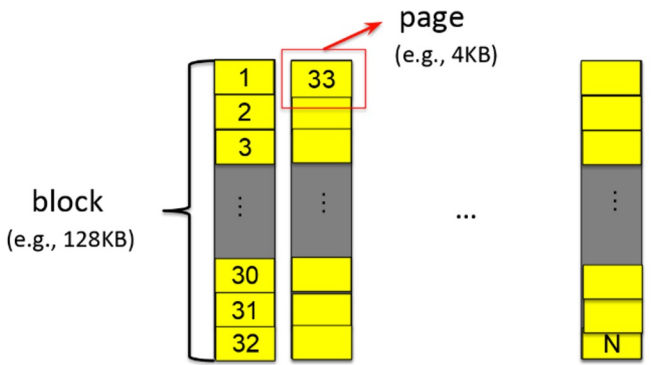
Control Area Network (CAN)

- CAN protocol uses two wires for communication between electronic control units (ECU)
- Messages are broadcast to all nodes on the network
- Nodes filter messages based on identifiers associated with messages
- CAN is accessible via on board diagnostics port (OBD-II) and used extensively in vehicles for sensors and control signals
- ECU firmware updates are often sent over CAN



Flash Memory

- Flash memory is broadly used as the external storage device for low-power embedded systems like ECUs
- Special nature of flash memory
  - Read/Write on pages, but erase on blocks
  - Erase-before-write
  - Out-of-place update
  - Limited number of program-erase (P/E) cycles



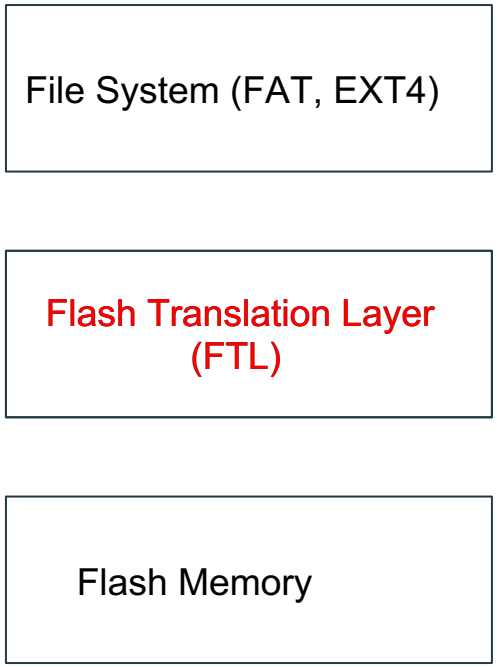
Special Functions Incorporated into Flash Storage Device

Garbage Collection: Blocks containing too many invalid pages will be reclaimed by copying valid data out of them, and the reclaimed blocks will be placed to free block pool to be reused

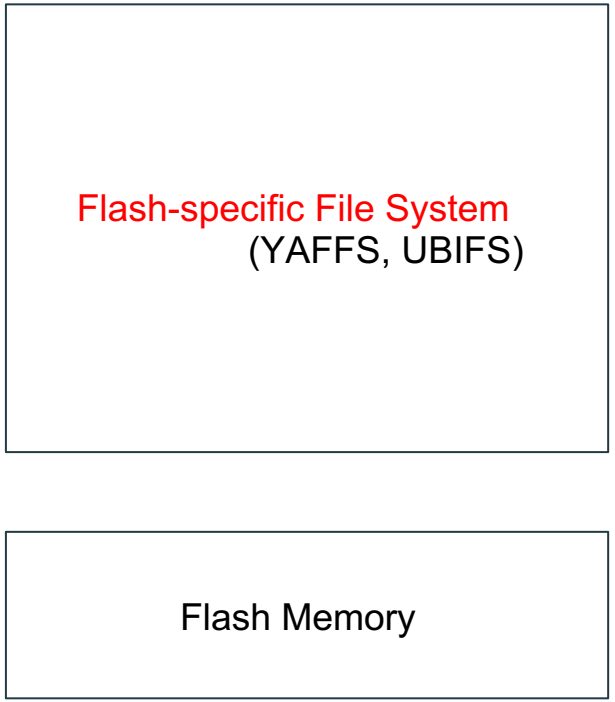
Wear Levelling: Distribute writes/erasures evenly across flash memory

Bad Block Management: A flash block may turn “bad” over time and cannot reliably store data. Bad block management typically introduces a bad block table to keep track of bad blocks. Once a block turns bad, it will be added to the bad block table and will no longer be used

How to Use Flash Memory



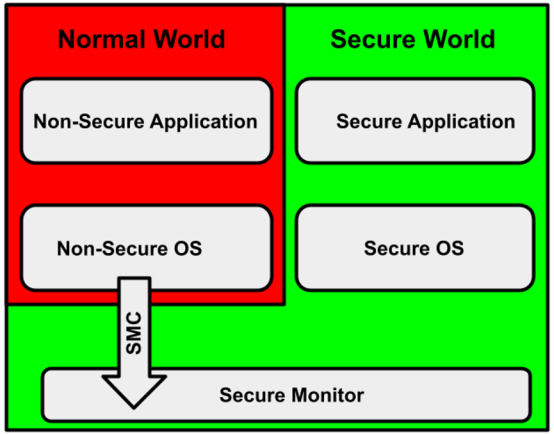
Method 1: FTL



Method 2: Flash File System

ARM TrustZone

- Many ARM processors, such as Cortex-A and Cortex-M CPUs used within automotive ECUs are ARM TrustZone enabled
- Two execution environments
  - Secure execution environment (secure world)
  - Non-secure execution environment (normal world)
- Each world operates independently when using the same processor and, switching between them is orthogonal to all other capabilities of the processor
- Secure world is isolated from normal OS, the OS-level malware cannot compromise secure world



Steganography

- A mechanism by which to hide some secret message inside of normal data/communications
- The secret message is embedded obscurely into original data or messages, such that it goes unnoticed
- Different from encryption, this is intended to conceal the fact that a secret message is being sent at all

System and Adversarial Model

System Model

- We consider a connected vehicle with multiple ECUs communicating via the CAN protocol. The ECU is assumed to be equipped with a NAND flash storage device (e.g., an eMMC, a microSD, etc.) on which the ECU firmware is stored
- The flash storage device is managed by an FTL, which provides a read/write interface to the ECU OS. The FTL is run on hardware isolated from the OS, so the computation performed by it is assumed to be secure
- Each ECU is assumed to be equipped with an ARM processor (Cortex-A or Cortex-M) with TrustZone enabled. The trusted world, running trusted applications (TAs) can communicate with untrusted client applications (CAs) running in the untrusted OS
- We assume the existence of a trusted in-vehicle computer (IDet) connected to the CAN bus, which performs intrusion detection and signal analysis to detect and localize adversarial ECUs
    - Note that IDet can communicate directly with the CA via the CAN bus

Attack Model

- We consider an adversary which can compromise the firmware of an ECU by injecting malicious code into the ECU OS
- Any detection and recovery mechanisms running in the ECU OS are averted since ECU is compromised
- The malware is detectable via intrusion detection of the the vehicle, as it must behave maliciously in order to take control of the vehicle

Assumptions

- The compromised ECU is not able to compromise the trusted applications (TAs) running in the TrustZone secure world, which is protected by the processor at the hardware level. This is a common assumption for TrustZone-based applications
- The compromised ECU is not able to hack into the FTL, which is isolated by the storage hardware and only presents a limited read/write interface
- Before the ECU is compromised, its firmware (OS) is assumed to be healthy
- The compromised ECU will not perform DoS attacks, e.g., blocking *regular* communication among CAN, CA, TA, and FTL

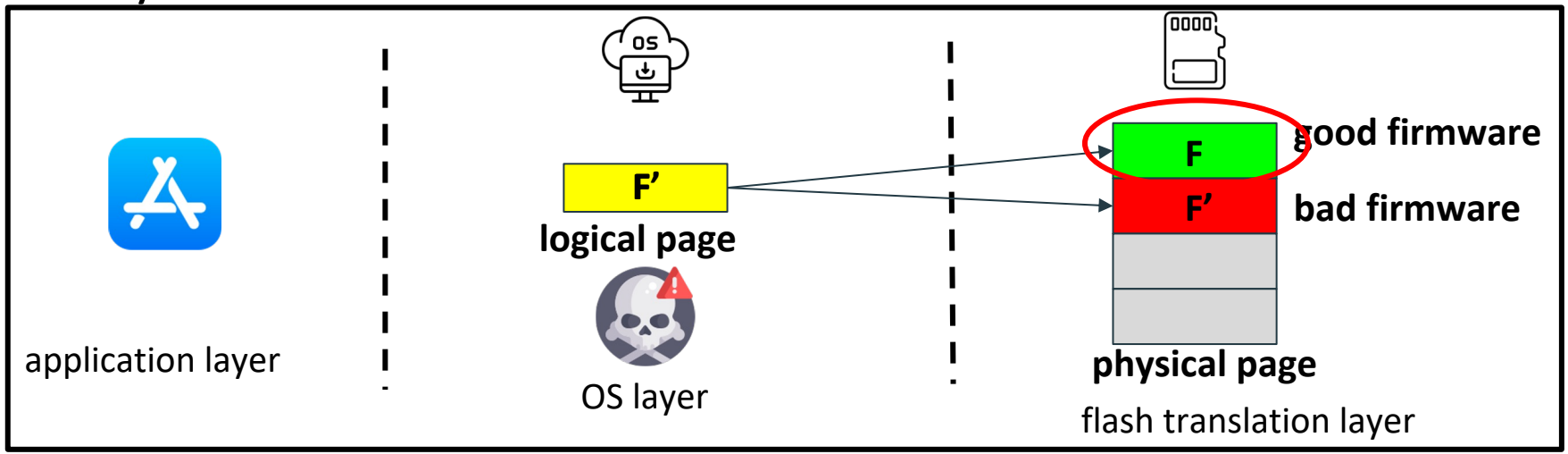
Design

Four Major Components

- Intrusion Detector (IDet): Running on top of trusted firmware in a secure node, which can communicate with the victim ECU via CAN network (trusted)
- Victim ECU
  - Client Application (CA): Running on top of the untrusted firmware which may be compromised (untrusted)
  - Trusted Application (TA): Isolated from the CA by TrustZone hardware (trusted)
  - FTL: Isolated from CA by storage hardware (trusted)

Key Ideas

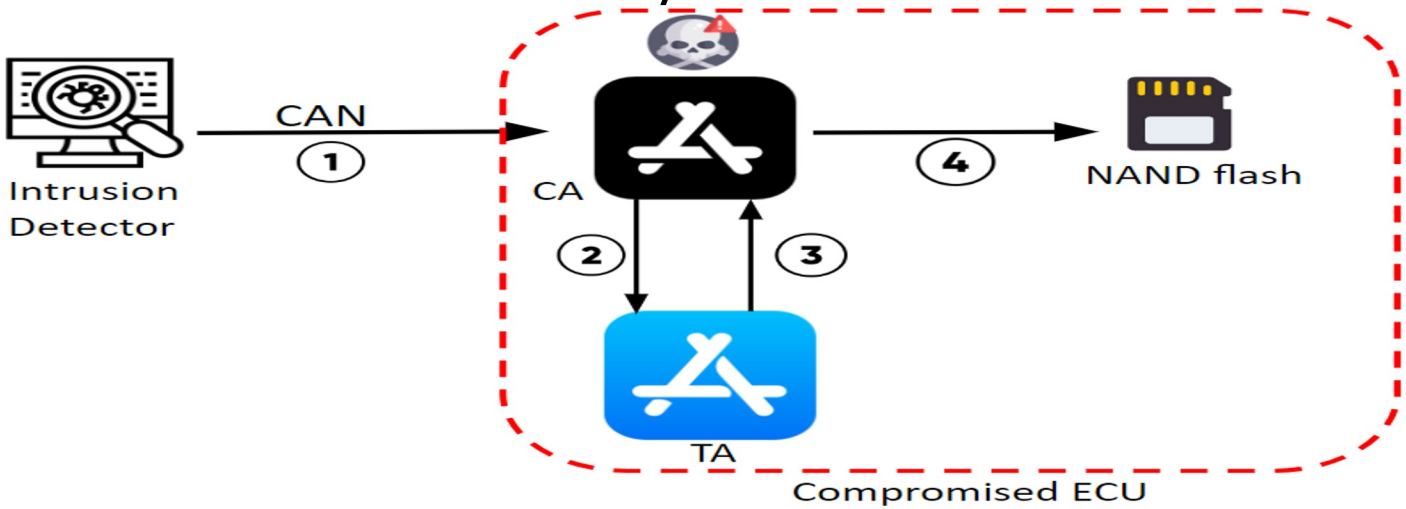
Idea 1: Leveraging the out-of-place update strategy of flash memory to restore the old version of firmware



- The good firmware is temporarily preserved due to the out-of-place update of flash storage
- The good firmware can be restored

Key Ideas (cont.)

Idea 2: Securely communicate between IDet and FTL to enable detection notification and recovery



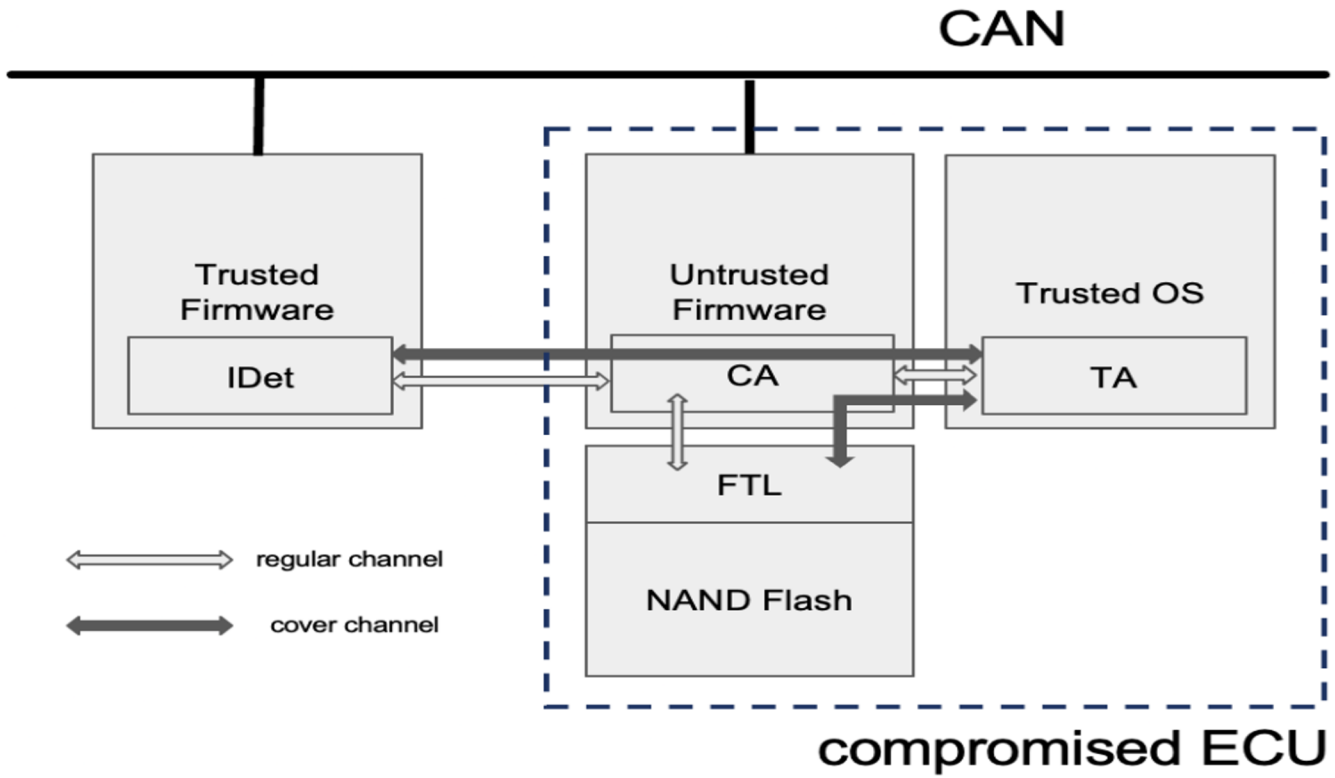
- For the FTL to initialize recovery, it must be notified securely of malware detection
- We hide the notification in regular communication via steganography, and use the TA as a decoy proxy

Key Ideas (cont.)

Idea 3: Enabling the restoration of the ECU firmware when the malware is still present

- It would be hard for the vehicle user to block the ECU malware once being detected
- Upon restoration, the FTL will block all the write requests from the upper layer, and this blocking operation will be canceled once the good firmware has be restored

Design Summary



Design Details

1: Cover communications via steganography

- Goal: Securely embed detection message β within regular communication α
- 2 Functions: π, a pseudo random permutation. ƒ, a pseudo random function
- k is the length of M_s, the steganographic message, and α
- l is the length of a secret message β
- S is the key length

$$\pi : \{0,1\}^s \times \{0,1\}^{\log_2 k} \rightarrow \{0,1\}^{\log_2 k}$$

$$f : \{0,1\}^s \times \{0,1\}^* \rightarrow \{0,1\}^s$$

Algorithm 1 SEncode

Input: β, α, key, counter
Output: M_s

$M_s \leftarrow \alpha$
stegKey $\leftarrow f_{\text{key}}(\text{counter})$
for $i = 0$ to $l - 1$ **do**
    $j \leftarrow \pi_{\text{stegKey}}(i)$
    $M_s[j] \leftarrow \beta[i]$
return M_s

Algorithm 2 SDecode

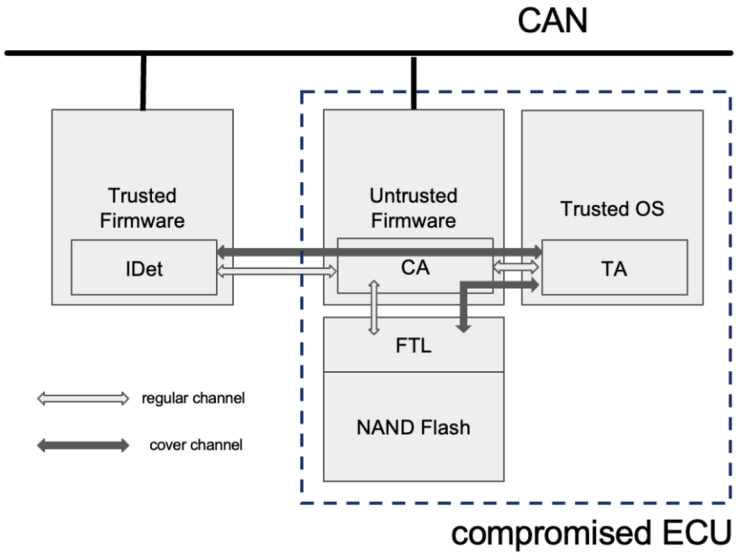
Input: M_s, key, counter
Output: β

stegKey $\leftarrow f_{\text{key}}(\text{counter})$
for $i = 0$ to $l - 1$ **do**
    $j \leftarrow \pi_{\text{stegKey}}(i)$
    $\beta[i] \leftarrow M_s[j]$
return β

Design Details (cont.)

1: Cover communications via steganography

- M_s is generated in IDet, using a key and counter shared by IDet and TA, and sent over CAN to CA
  - CA forwards this to TA
- TA generates M_{s1} using a key and counter shared between CA and FTL
  - TA sends M_{s1} to CA, indicating it should be written to FTL
- FTL decodes M_{s1}, starting recovery if necessary



Design Details (cont.)

2: Firmware restoration

- Challenge 1: The old firmware must still be present in a recoverable manner on the storage device
  - Use out-of-place updates
  - Save the old mappings during updates to reserved area
  - Block garbage collection on old firmware blocks
- Challenge 2: The old firmware should be restored quickly to the correct location
  - Restore the saved mappings during recovery

Design Details (cont.)

3: Malware removal

- The malware may still be running on the CPU and contained in the ECU memory even if the ECU firmware is restored to the good version

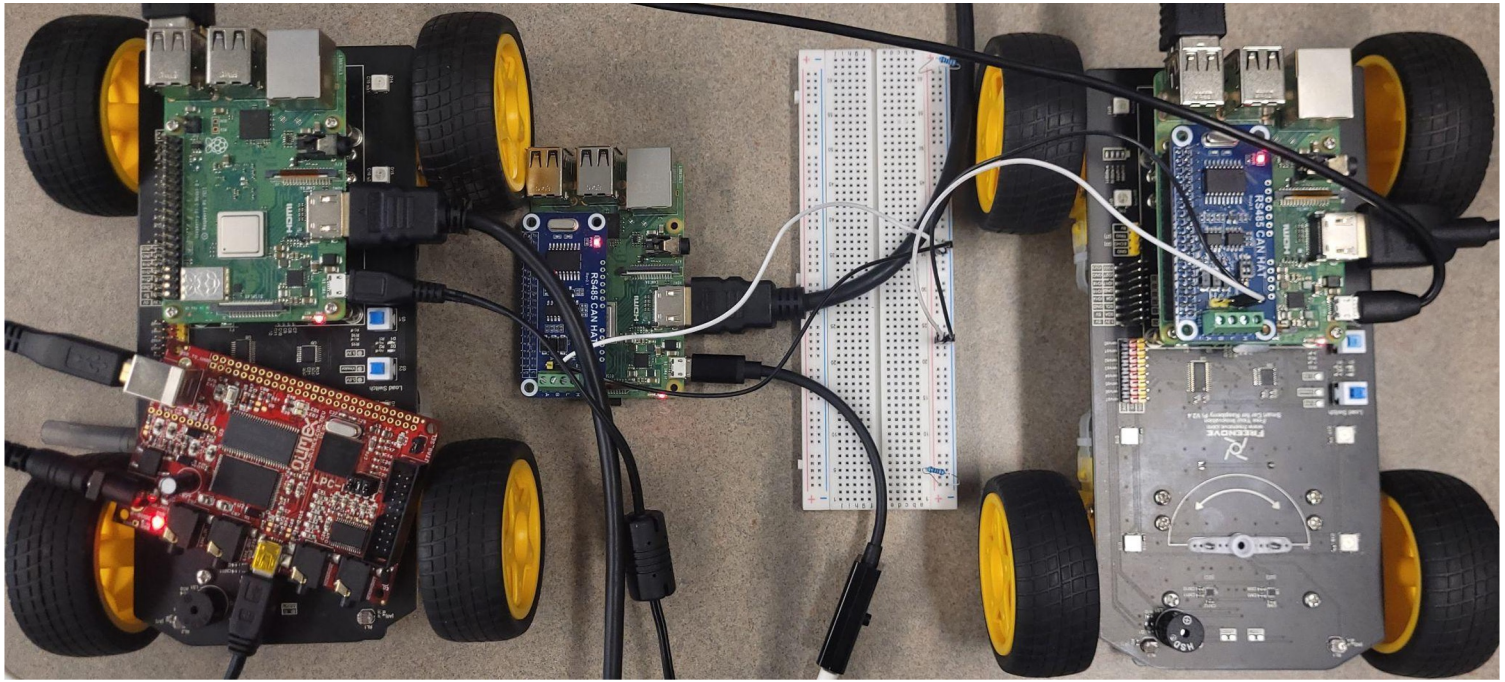
- Our solution
  - Disable writes on FTL to prevent rewriting the malware
  - Reboot ECU to flush malware from memory
  - Enable writes again via notification from the restored OS

Implementation

Implementation (cont.)

- Raspberry Pi 3B+
  - Used as IDet and ECUs
  - 4GHz 64-bit quad-core ARM Cortex-A53 CPU, 1GB LPDDR2 SDRAM)
- LPC-H3131
  - Used as the external storage of ECU
  - ARM9 32-bit ARM926EJ-S, 180Mhz, 32MB of SDRAM, and 512MB NAND flash
- RS485 CAN hat
- Firmware modification
  - Modified OpenNFM and ported it to LPC-H3131

Implementation (cont.)



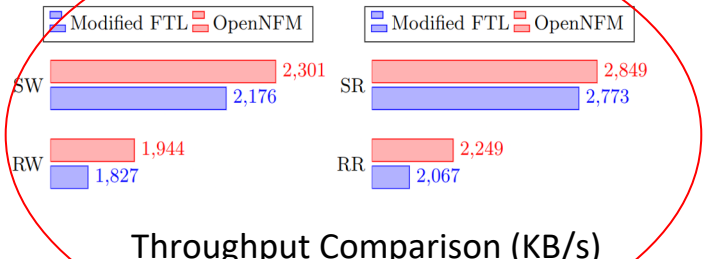
Evaluation

Evaluation

- Performed timing evaluations in terms of timing in IDet, TA, and FTL
  - After malicious activity is detected, the good firmware can be quickly restored
- Performed throughput evaluation with FIO, measuring the impact on normal use
  - Sequential and random write/read were measured
  - Average throughput difference of 5.7%

IDet Time (s)	TA Time (s)	FTL Time (s)
.0012988	7.1065939	1.4264553

Average time in IDet, TA, and FTL



Throughput Comparison (KB/s)

Conclusion

- We have designed a new framework for connected and autonomous vehicles to defend against the ECU code injection attacks, by rolling back the compromised ECU firmware to a good prior state
- We take advantage of various existing hardware features equipped with the ECU to securely manage and efficiently perform the recovery process
- We have implemented a prototype for the proposed framework and demonstrated its effectiveness at performing real-time recovery in a simulated in-vehicle testbed

Thanks!