# CAROT: A Secure RISC-V ECU with Trusted Execution and Moving Target Defense

Ahmer Raza, Zhenkai Zhang
Clemson University, Clemson, SC, USA
{araza, zhenkai}@clemson.edu

***Abstract*—The modern vehicle implements Electronic Control Units (ECUs) for various safety-critical features, such as engine management, safety systems, and advanced driver assistance systems. However, ECUs and their interconnecting Controller Area Network (CAN) bus have not been developed from a security-first perspective, and are susceptible to various attacks. In particular, an attacker may use remote or physical attack vectors to breach an ECU, thus gaining access to the CAN bus and allowing for attacks on other ECUs within the In-Vehicle Network (IVN). To mitigate these issues, we present CAROT: a secure, efficient, open-source ECU based on RISC-V. We use Trusted Execution Environments (TEEs) and Instruction Set Randomization (ISR), a Moving Target Defense (MTD) technique, to protect against common code injection and operating system attacks that ECUs face. We implement our design on real hardware via an FPGA-based RISC-V prototype, and perform evaluations in a simulated IVN environment using CARLA. We report that our ECU provides security against our threat model while only incurring a 1.05% response latency overhead.**

***Index Terms*—Automotive security, electronic control unit (ECU), moving target defense (MTD), RISC-V, trusted execution environment (TEE)**

## I. INTRODUCTION

Cars are no longer simple mechanical devices. The average automotive vehicle contains a myriad of complex features, including safety systems, infotainment, engine management, traction control, and Advanced Driver-Assistance Systems (ADAS) such as Adaptive Cruise Control (ACC) and Lane Keeping Assist (LKA) [1]. The implementation of these modern features requires powerful processing units that can handle a massive amount of information. Most vehicles modularize these features into hundreds of Electronic Control Units (ECUs), each responsible for a specific task. The number of ECUs in the average car is progressively increasing as manufacturers incorporate increasingly advanced features into their vehicles. Furthermore, the development of autonomous vehicles requires even more sophisticated and powerful ECUs. Depending on their criticality, ECUs must satisfy strict real-time constraints to ensure the stability of the entire vehicle. To implement these advanced features, ECUs have become progressively similar to regular computing systems, having powerful processing units and being able to handle a massive throughput of information.

To guarantee such efficiency and reliability while still reducing manufacturing costs, ECUs within a car are interconnected using shared buses. The combination of a vehicle's ECUs, interconnect buses, and additional sensor networks is called the In-Vehicle Network (IVN). The most common bus used to link ECUs together is known as the Controller Area Network (CAN) bus, an industry-standard protocol and the one we exclusively consider in this paper.

In this context, ECUs and the IVN in modern vehicles have not been developed from a security-first perspective, and do not provide the basic security requirements of confidentiality, authenticity, availability, integrity, or non-repudiation [2]. As a result, various existing attacks have been found to affect ECUs, and many ECU-specific vulnerabilities have also been found [2]–[5]. In particular, the CAN protocol is highly susceptible to masquerade and packet injection, where an external attacker can simply connect to the physical CAN bus and compromise the system. Moreover, ECUs implemented in practice do not protect against remote security breaches or intrinsic vulnerabilities in their logic. This is alarming, especially considering the many safety-critical features managed by such ECUs. Even if an attacker does not gain access to the highly important ECU, they can still remotely manipulate a breached ECU to inject messages on the CAN bus, easily compromising the whole IVN [4].

Significant work has been done to mitigate the issue of unauthorized CAN bus access [6]–[19]. However, most approaches rely on Intrusion Detection System (IDS) schemes or authentication methods. IDS solutions require a dedicated CAN node and constant monitoring, and only detect attacks post-compromise, while authentication-based schemes require sender and receiver authentication calculation, incurring a large (and sometimes unnecessary) overhead. Further, the issue of protecting ECU logic itself from exploitation—whether because of vulnerabilities in ECU code or general attacks on the application—remains largely unexplored.

In order to address these shortcomings, we propose CAROT (Car Root-of-Trust), a secure, open-source ECU based on the RISC-V platform. Compared to previous work, we adopt an end-to-end security approach. Our goal is to block unauthorized access to the CAN bus through ECUs from the outset, and to mitigate ECU exploitation even if an attacker gains access to the bus. The CAROT platform employs Trusted Execution Environments (TEEs) and Moving Target Defense (MTD) to prevent the outlined threats against IVNs. We hypothesize that our system provides protection against automotive code injection and local Operating System (OS) attacks, with minimal impact on the performance of the ECU. Our paper makes the following contributions:

- We design a fully open-source ECU platform using RISC-V, incorporating the concepts of the Instruction Set Randomization (ISR) MTD and TEEs to deter attacks on the ECU.
- We implement our platform on a real RISC-V System-on-Chip (SoC), including our ISR implementation, full TEE support, and secure CAN bus access.
- We evaluate our implemented platform by simulating a real vehicle IVN, and compare its security and performance with a baseline ECU. Specifically, we demonstrate code injection and OS attacks to be ineffective against our ECU implementation, and compare its response time latency to that of the baseline ECU.

The paper is outlined as follows. Section II discusses fundamental background knowledge behind our design. Section III presents the threat model we consider when designing our platform. Section IV describes the design of the CAROT platform, including system assumptions and our additions. Section V gives an overview of our implementation of the CAROT platform on real hardware. Section VI presents our security and performance evaluations of the implemented platform. Section VII discusses prior related work in the area of IVN security. Finally, Section VIII ends the paper with a summary and concluding remarks.

## II. BACKGROUND

In this section, we provide the prerequisite background information on vehicle IVNs, TEEs, and MTD. We discuss the implementation and shortcomings of real IVNs; the trusted computing paradigm; TEEs, which are used to provide a trusted environment; and MTD techniques, including ISR, which is our MTD of choice.

### A. Interconnected ECUs

ECUs are typically implemented using microcontrollers and are interconnected via communication buses, the most common being the CAN bus: a robust, general-purpose, serial communication protocol designed to efficiently broadcast small messages over a shared bus. Along with sensors and peripherals, these interconnected ECUs form the IVN, which controls all aspects of a vehicle's operation.

From a security perspective, the CAN bus is highly vulnerable to attacks due to its lack of built-in security mechanisms. The protocol does not provide encryption, authentication, or integrity checks, making it susceptible to eavesdropping, message injection, and replay attacks [2]. For example, an attacker with physical access to the CAN bus can easily inject malicious messages, spoof legitimate ECUs, or disrupt communication between ECUs. Additionally, the broadcast nature of the CAN bus means that any compromised ECU can send messages to all other ECUs, potentially taking control of critical vehicle functions. ECUs themselves are also vulnerable to attacks. Many ECUs run legacy software with known vulnerabilities, such as buffer overflows or insecure firmware updates. Attackers can exploit these vulnerabilities to gain unauthorized access to the ECU, manipulate its behavior, or extract sensitive information [3], [4]. Furthermore, the increasing complexity of ECU software and the use of third-party components introduce additional attack surfaces, making it difficult to ensure the security of the entire IVN.

### B. Trusted Execution Environments

The classic Rich Execution Environment (REE) paradigm involves user-mode code running on a kernel, which further uses firmware that interfaces with hardware directly. This paradigm is host to various vulnerabilities, stemming from the execution of untrusted code and a lack of sufficient memory protection mechanisms. Trusted computing is a technique developed to help mitigate these problems by providing secure computation, privacy, and data protection [20]. A practical implementation of trusted computing is the Trusted Execution Environment (TEE) paradigm. A TEE is an isolated processing environment that runs in parallel with the REE on the same SoC. Its core function is to protect security-sensitive applications from the potentially dangerous main OS and other REE software. The TEE approach attempts to minimize the Trusted Computing Base (TCB), which is the class of software that is safety-critical to the operation of the TEE, making it easier to verify and trust than the entire platform. A TEE is self-contained; it operates without relying on the external, untrusted operating system for its core security functions, thereby ensuring that the security of the enclave is separate from the vulnerabilities in the REE [21].

To achieve isolation between the TEE and the REE, the system uses a hardware-level Root-of-Trust (RoT) to initialize the secure TEE environment. During runtime, this environment is monitored by a privileged component called the separation kernel, which controls all transitions between the untrusted REE and the trusted TEE. The architecture of the TEE itself generally falls into two categories: trusted world approaches and enclave approaches.

Keystone is an open-source TEE framework that implements the latter enclave approach on RISC-V architectures [22]. Its goal is to only provide the basic primitives for architecting TEEs, rather than a complete, full-stack solution like Intel SGX or ARM TrustZone [23], [24]. Keystone relies on standard RISC-V features, particularly Physical Memory Protection (PMP), to enforce isolation from the coexisting REE. RISC-V defines three privilege levels: user-mode (U-mode) for applications, supervisor-mode (S-mode) for the OS kernel and system programs, and machine-mode (M-mode) for firmware. Keystone uses these levels to structure the enclave, which consists of the enclave application (eapp) in U-mode, the enclave runtime kernel (called Eyrie) in S-mode, and the security monitor (SM) in M-mode. The resulting additions to the TCB are thus minimal and customizable, encompassing only the hardware RoT, the small SM, and the modular Eyrie kernel, which can be configured to include only necessary features. We describe the realized implementation of Keystone further in Section V-A, when we discuss our integration of Keystone in CAROT.

### C. Moving Target Defense

MTD is a technique that constantly changes the attack surfaces of a system. It increases the uncertainty and complexity of the system, making it difficult for bad actors to exploit vulnerabilities or perform attacks [25], [26]. The end result is the attacker wasting effort on trying to learn inner details about a system, while not gaining any useful information. Implementations of MTD thus usually involve randomization of some element of the system.

Most operating systems already implement the Address Space Layout Randomization (ASLR) MTD in their virtual memory scheme [27]. ASLR is a technique that attempts to guard against buffer overflow attacks by randomizing the virtual memory addresses of certain components, such as the base executable, shared libraries, and the stack. However, common implementations of ASLR—namely, PaX ASLR on Linux and Dynamic Base/High-Entropy ASLR on Windows—have been circumvented with relative ease, making the system ineffective in practice [27]–[30]. Moreover, ASLR does not prevent memory corruption or code injection itself; it only hinders the reliability of such attacks. Platform-specific bypasses may still allow attackers to execute injected or reused code despite ASLR.

ISR is an MTD developed for protection against general code injection attacks [31]–[33]. The core idea behind ISR is to randomize the instruction encoding of a program such that injected malicious code, which assumes a standard instruction set, fails to execute correctly. This is achieved by encrypting or transforming the binary instructions at load time and decrypting them at runtime using a corresponding key. Traditional ISR implementations can apply randomization at different granularities, including whole binaries, function-level transformations, or instruction-by-instruction modifications. These transformations ensure that an attacker who injects raw shellcode or attempts to reuse existing code will be unable to execute meaningful instructions unless they possess the decryption key.

ISR is implemented as a three-part system: a randomizing element, an appropriate execution environment, and the binary loader [32]. The randomizing element is a program that applies the chosen transformation to the program binary after compilation. The execution environment, implemented within the processor if possible, reverses the operation performed by the randomizing element; it is subject to strict performance constraints, as even a small overhead adds up quickly when considering large programs. Finally, aside from all of the regular program initialization and loading requirements, the binary loader is responsible for extracting and providing the key to the execution environment at runtime.

## III. Threat Model

Our threat model considers two classes of attacks: (i) code injection attacks targeting vulnerabilities due to insecure ECU logic, and (ii) attacks originating from root access to the operating system running alongside the CAROT platform. OS-launched attacks may arise from remote exploitation of network-facing services in the telematics ECU or from physical access to the device. A representative attack chain could proceed as follows. First, a remote adversary compromises the telematics ECU through a vulnerability in its software stack, thereby gaining access to the IVN. The attacker can then transmit crafted CAN messages to a critical ECU containing a buffer overflow vulnerability. By exploiting this flaw, the adversary triggers arbitrary code execution in the ECU runtime.

In line with the TEE paradigm, we assume the operating system to be fully untrusted: CAROT does not impose restrictions on the CAN bus, other ECUs, or sensors within the IVN. This reflects real-world IVNs, which present a wide variety of attack surfaces and often use legacy code that lacks modern compiler-level protections [2]–[5].

## IV. Design

Our priorities when designing CAROT are to provide fully open-source components across the software and hardware stack, and to minimize the TCB. Minimizing the TCB is crucial as it represents the set of components that must be trusted for system security. A smaller TCB reduces the attack surface and verification complexity. Our design controls TCB growth by ensuring additions to critical components in hardware/firmware are minimal. Although it is also desirable to keep the runtime small, its size is less critical than that of the firmware. Moreover, under our model, reducing the runtime size is severely limited by the large footprint of essential CAN and peripheral drivers.

To achieve these goals and protect against our threat model, the design of CAROT consists of four components across the three privilege modes.

1) *Trusted hardware*, with a secure boot procedure and ISR.
2) *Machine-level firmware*, with a trusted separation kernel that interfaces the REE, TEEs, and hardware peripherals. This component comprises most of the TCB.
3) *Supervisor-level software*, consisting of an untrusted OS running alongside a TEE runtime. The runtime may contribute to some of the TCB.
4) *User-level software*, containing the programmed ECU logic.

We design our platform around the RISC-V open-source Instruction Set Architecture (ISA). Our high-level design is shown in Figure 1 and is described in detail below.

### A. Trusted Hardware

CAROT assumes the underlying hardware is trusted. This includes the Central Processing Unit (CPU) cores themselves, as well as peripherals and hardware systems within the SoC. CAROT implements ISR within the CPU, which is used by the TEEs to decrypt ECU logic code. The implemented ISR scheme is efficient, which prevents timing issues within the pipeline and overall CPU slowdown. The platform also provides a root-of-trust that performs a secure boot procedure within the default boot ROM logic, which validates digital signatures of further boot stages, ensuring only trusted code is booted. Considering these additions, the hardware platform

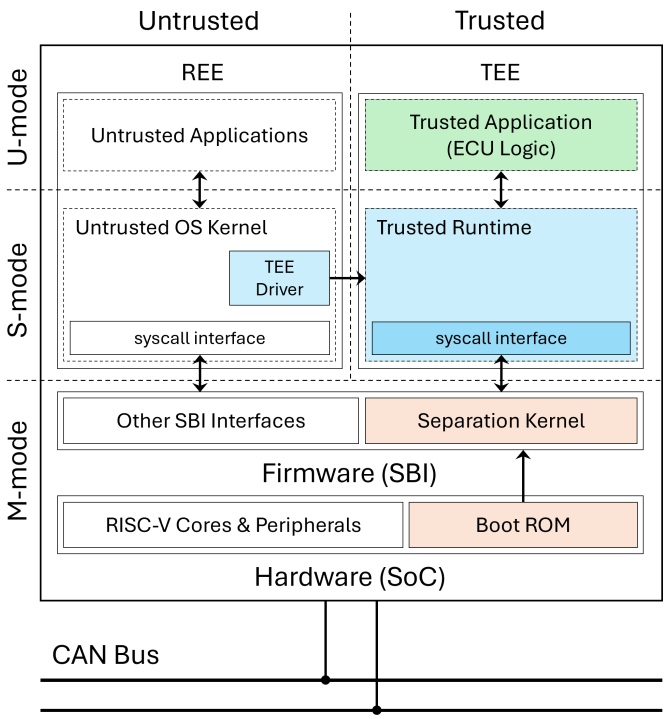



Fig. 1. CAROT RISC-V ECU. High-level design, describing the software TEE (U-mode and S-mode), and hardware/firmware (M-mode) components, with additions highlighted.

we choose is RISC-V. Being open-source, many accessible and easily modifiable CPU implementations exist, making ISR and prototyping much easier than other platforms, and making a customized secure boot procedure simple to implement. Furthermore, being an open platform, RISC-V allows for greater transparency and verifiability of the hardware, upholding our assumption of trusted underlying hardware.

Within the SoC, CAROT implements a CAN bus controller interface for use by the ECU logic code. The controller may be contained in the SoC, or it may be an external device connected to the SoC through Serial Peripheral Interface (SPI). Importantly, through the separation kernel associated with the TEE, the CAN bus controller is only exposed to the secure world (TEEs). No non-secure code in the REE is able to access the CAN bus.

### B. Firmware

The default firmware used by CAROT is hardware-specific, but at the very least provides syscall support. The separation kernel is built into the firmware, and is used to launch TEEs and handle inter-TEE and TEE-REE communication. It runs alongside regular RISC-V Supervisor Binary Interface (SBI) interfaces, and provides an additional interface for TEE-specific utilities. In particular, the separation kernel manages access to the I/O peripherals and protects their virtual memory spaces, which guarantees secure and exclusive access to the sensors and CAN bus. Depending on the implementation, the firmware is either the complete TCB or is a major component of the TCB. Nevertheless, firmware usually has the largest impact on TCB size and must be very carefully implemented.

### C. Software

As our threat model assumes the OS is untrusted, we do not place any restriction on the REE, so it may be anything, including a bare-metal system, a Real-Time OS (RTOS), or a general-purpose OS. However, the REE must include some syscall interface to the separation kernel. It also must allow CAROT to run a TEE driver, which is used to set TEE parameters and request the TEE to be launched. The TEE itself consists of a supervisor-mode runtime and a user-mode application which, in our case, encapsulates ECU logic. The choice of TEE runtime is more restricted, but any lightweight runtime or kernel will work if it can be modified to provide an interface to the separation kernel. Within the supervisor runtime, CAROT also provides drivers to interface the CAN bus managed by the separation kernel. While the user-mode TEE application does not contribute to the TCB, the supervisor-mode runtime may. If the runtime is included as part of the TCB calculation, the effect on TCB size may be far more substantial than that of the firmware, depending on the specific runtime used.

Even if a malicious actor cannot gain control of the vehicle through a code injection attack, they can still crash the system by inducing an error state, whether through a failed code injection attack attempt or some other attack vector. To mitigate this issue, our software component also includes a fail-safe automotive controller. When the system enters an error state, the runtime handles the exception by jumping to this default fail-safe controller, implemented as a simple “stop” or “pull over” sequence.

### D. Analysis

Together, these four components (hardware, firmware, and U-mode and S-mode software) provide comprehensive protection against all threats outlined in our threat model, while still retaining utility and performance. Figure 2 illustrates the resulting software flow of a CAROT ECU. The `launch_ecu` program—running on the host OS—sends a TEE launch request to the TEE driver. The driver transfers control to the TEE initialization procedure, which loads the ECU logic and jumps to it, enabling ISR appropriately. The ECU logic then communicates with the CAN controller driver, which directly interfaces the physical CAN controller. If the ECU enters an error state, the runtime jumps to the fail-safe controller, which then takes over communication with the CAN bus.

The project most comparable to ours is the TEECheck platform, which uses TEEs to provide a CAN bus authentication scheme [19]. Aside from the CAN bus access protection offered by both TEECheck and CAROT, our platform provides additional mitigation of vulnerabilities in ECU logic. Further, being open-source, CAROT allows for easier verification of trustworthiness across hardware, firmware, and software.

Consider an ECU running CAROT, executing its designated ECU logic within a TEE. The TEE ensures that the ECU logic

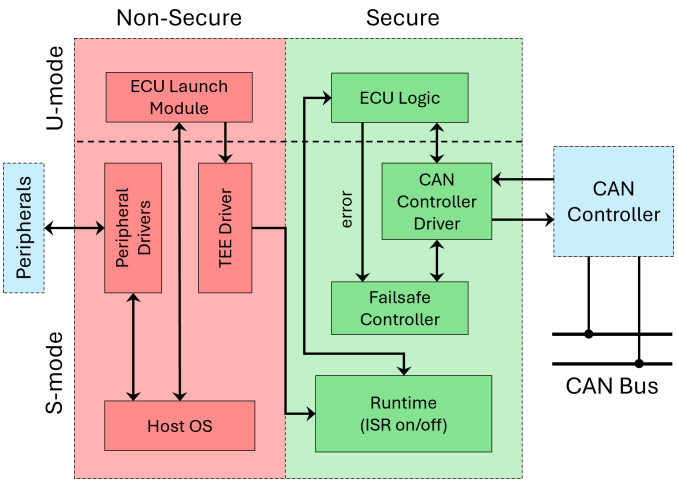



Fig. 2. Software flow diagram of ECU logic. Interface between non-secure REE and secure TEE, as well as between TEE and CAN controller.

remains isolated from the host operating system, meaning that even if the OS is compromised, the critical ECU operations are unaffected. This isolation prevents OS-launched attacks, such as privilege escalation or malware infiltration, from impacting the ECU's secure execution. Furthermore, assume the ECU logic has, for instance, a buffer overflow vulnerability, using which an attacker can launch a code injection attack. Even if the attacker manages to exploit this vulnerability, traditional code injection attacks are rendered ineffective due to the presence of the ISR mechanism. The attacker does not know the particular instruction set used by the ECU, and thus cannot execute meaningful code. Additionally, any attack that induces an error state in the vehicle can be handled smoothly by the fail-safe controller.

## V. Implementation

We implement the CAROT platform around the Rocket CPU core, using an open-source RISC-V SoC [34], [35]. The SoC comprises basic I/O and Dynamic Random Access Memory (DRAM) controllers, as well as the single-core 50 MHz Rocket CPU. We choose the Rocket CPU primarily because it uses an in-order pipeline, making ISR implementation easier. We prototype CAROT on an Arty A7-100T FPGA (Field-Programmable Gate Array) board running the SoC and our platform. Our implementation adds three major components to the base SoC and firmware:

1) Integration of Keystone enclaves, including secure boot and Keystone submodules.
2) Implementation of hardware ISR.
3) Addition of CAN bus utilities, including a physical controller interface in the SoC and Keystone in-enclave SPI and CAN drivers.

### A. Keystone Integration

The Keystone framework is a TEE implementation for RISC-V that provides a customizable, extensible, open-source alternative to existing TEE implementations [22]. It provides the capability to run partitioned enclaves (TEEs) through a provided separation kernel. To achieve this, Keystone adds two main components to a RISC-V platform: an M-mode Security Monitor (SM), which serves as the separation kernel; and an S-mode runtime that runs on top of the SM, inside of an enclave (TEE). Technically, the SM is the only component Keystone adds, since the runtime is optional. However, practically, using a different runtime than what Keystone provides involves resolving the many incompatibilities and dependency issues that may arise from porting the runtime. Keystone also provides an SDK for creating eapps (enclave apps; i.e., the actual U-mode application a user wishes to run) and a Linux kernel driver for requesting enclave launches from the SM.

The Keystone platform is designed to be portable to an arbitrary RISC-V system, but customization is still required to port Keystone. Prior to loading the SM, inside the boot ROM, we implement the secure boot procedure that Keystone requires. The boot procedure consists of (1) generating a public/private device keypair, (2) measuring (hashing) the SM, (3) hashing the SM hash alongside the device secret key, (4) generating a public/private keypair for use by the SM, and (5) endorsing (signing) the step (3) hash and SM public key. We then place the device keypair, SM keypair, SM hash, and signature in memory for the SM to access. In total, we pass 352 bytes of data to the SM.

As Keystone uses OpenSBI v1.1 while our platform uses OpenSBI v1.6, we modify both the SM and OpenSBI before integrating the two. Importantly, Keystone must update Physical Memory Protection (PMP) settings in response to Inter-Processor Interrupts (IPIs), and the IPI handler has changed across these versions, which we take into consideration. Aside from minor calling convention differences, there are not any major obstacles we encounter while porting the SM. Since the Keystone runtime and user-mode SDK are fully managed by the SM, they do not require any changes aside from the security additions we provide.

### B. CAN Bus Access

Inside the SoC, we provide a Xilinx SPI master bus with which an external CAN controller/transceiver combination can be controlled. Specifically, alongside our FPGA, we use the MCP25625 integrated CAN controller/transceiver chip. We ensure the CAN controller is only exposed to Keystone TEEs and not the host OS by assigning a property `secure-status="okay"` to the device tree node that describes the controller. When parsing the device tree, the Keystone SM will ensure that only TEEs can access the CAN bus, and completely conceal the device's presence from the host OS. The Keystone runtime and SM provide experimental support for Memory-Mapped I/O (MMIO) peripherals; however, no drivers are provided. To this extent, we have written both SPI and CAN drivers for in-enclave CAN bus access. While the SPI driver is minimal, the CAN driver fully supports the CAN 2.0B protocol.

Keystone Enclave does not provide any timer interface to the runtime. Additionally, there is no interrupt controller

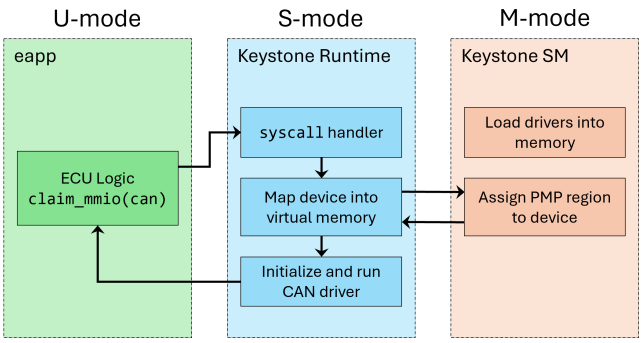



Fig. 3. CAN bus access through Keystone. Propagation of the device claim syscall through Keystone's components.

that abstracts hardware interrupts and allows for interrupt callbacks. Hence, our CAN bus drivers operate asynchronously and are forced to use polling to check for incoming CAN packets. Access to the CAN bus is outlined in Figure 3. During the SBI boot stage, the SM uses the platform's device tree to load appropriate drivers into memory. Later, during enclave execution, the eapp requests access to the CAN bus and respective driver through a Keystone runtime syscall. The runtime creates an abstract virtual memory region that allows the driver to manipulate registers on the hardware. The SM allocates a PMP register for this virtual mapping, ensuring complete memory protection. The runtime then initializes and runs the CAN bus driver, which was loaded by the SM prior. Finally, control is returned to the eapp, which interfaces with the driver to send and receive CAN packets.

### C. ISR Implementation

We modify the Rocket CPU to provide ISR, as shown in Figure 4. Firstly, we provide two new RISC-V machine-level Control and Status Registers (CSRs): the 1-bit `misr` register, which controls whether ISR is enabled; and the 32-bit `mkey` register, which stores the ISR key with which to decrypt a randomized instruction. Since these are M-mode registers, only M-mode code is permitted to modify these registers. During the instruction decode stage after an instruction is received, if ISR mode is enabled, the CPU performs the appropriate decryption using the key stored in `mkey`. In order to handle privilege mode switches, we only perform ISR decryption when the CPU is in U-mode. This guarantees that non-ISR instructions will not be erroneously decrypted. In our current implementation, we perform a simple XOR operation to both encrypt and decrypt instructions. RISC-V supports compressed 16-bit instructions that are later decompressed by the CPU in the instruction decode stage. We handle these half-length instructions by simply using the lower 16 bits of the 32-bit key, both when encrypting and decrypting. This provides adequate security under our assumptions, while still adhering to clock constraints and ensuring there are no timing issues. In order to implement separate ISR schemes, such as block ciphers or permutation methods, we can simply modify the decryption module within the CPU.

As the primary purpose of our ISR implementation is to provide protection within an enclave, we wrote a utility that encrypts an ELF executable using a specified key after compiling the program. Using a linker script, we store the encryption key in a separate section within the ELF file. To prevent attackers from simply reading the key from the file and decrypting it, we encrypt the ISR key using an RSA public key provided by the Keystone runtime. After loading an eapp, the runtime then decrypts the ISR key and stores it in `mkey`. Since the CPU checks for U-mode before performing ISR decryption, we only need to encrypt the eapp; not the entire Keystone runtime.

In order to be able to enable and disable ISR, we implement two syscalls in the Keystone SM: `enable_isr` and `set_isr_key`, which set the `misr` and `mkey` registers, respectively. Through the SM-SBI interface, the Keystone runtime can treat the SM syscall as an OpenSBI syscall. The Keystone runtime handles all ISR enabling and disabling during runtime-to-eapp and eapp-to-runtime context switches. We do not provide an interface for eapps to control ISR directly, which allows for complete abstraction of ISR for the eapp.

### D. Impact on TCB

The resulting increases in the size of the TCB and hardware due to our additions, measured in lines of code (LOC), are broken down in Table I and Table II. The largest size increase in the root-of-trust is in the boot ROM, with 18.86% additional lines of code. This is explained by the incorporation of the ED25519 and SHA libraries, which are needed for hashing, signing, and encryption/decryption in the secure boot procedure. The most significant percent increase in TCB size is caused by the addition of our complex SPI and CAN drivers, while the largest LOC impact on TCB comes from the firmware.

TABLE I
LOC BREAKDOWN FOR ROOT-OF-TRUST AND OTHER HARDWARE. INCREASE IN LOC AFTER IMPLEMENTING CAROT.

| | Rocket (ISR) | Boot ROM | Total |
|---|---|---|---|
| Base | 10,450 | 22,000 | 32,450 |
| Additional | 40 | 4,150 | 4,190 |
| Increase | 0.38% | 18.86% | 12.91% |
| Total | 10,490 | 26,150 | 36,640 |

TABLE II
LOC BREAKDOWN FOR TCB. INCREASE IN LOC AFTER IMPLEMENTING CAROT.

| | SM | OpenSBI | Runtime | Total |
|---|---|---|---|---|
| Base | 7,080 | 21,200 | 10,050 | 38,330 |
| Additional | 140 | 20 | 1,200 | 1,360 |
| Increase | 1.98% | 0.09% | 11.94% | 3.55% |
| Total | 7,220 | 21,220 | 11,250 | 39,690 |

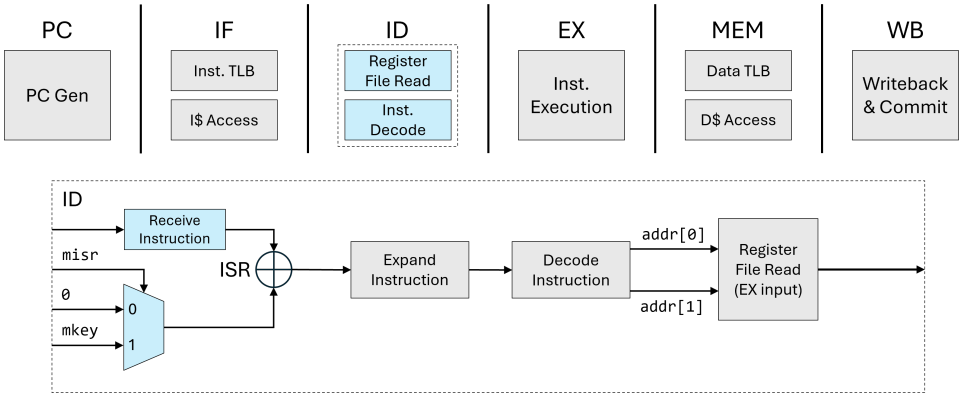



Fig. 4. Rocket CPU pipeline and ISR implementation. Additions within the ID stage in the Rocket pipeline to implement ISR.

## VI. Evaluation

We evaluate the implementation of the CAROT platform by emulating a real ADAS ECU. Our implementation of the ADAS consists of LKA logic: given lane line information as an input, the ADAS LKA system controls the steering angle of the vehicle, ensuring the vehicle is always centered in its lane. In order to compare against a baseline, we also implement a baseline CAROT ADAS ECU, with ISR disabled and ECU logic running on the host OS instead of within a Keystone enclave.

### A. IVN Setup

To emulate a real vehicle, we make use of the open-source CARLA driving simulator [36]. Our CARLA simulation consists of a vehicle with an RGB camera. We implement a Computer Vision (CV) algorithm in OpenCV to extract lane line curves from the RGB camera output. The ADAS is implemented as a CAROT ECU connected to a CAN bus. To interface the CARLA simulator with the physical ECU, we create a CAN bus bridge and connect it to the ECU CAN bus, simulating a real vehicle IVN. The lane line curve information is given to the LKA ECU, and the ECU returns corresponding steering angle adjustments. The LKA logic within both testbed ECUs is implemented as a simple proportional controller. This setup is shown in Figure 5.

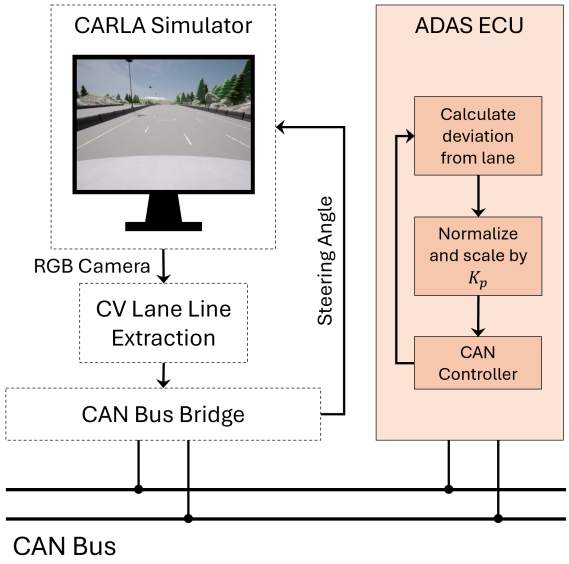



Fig. 5. Experimental setup. Implementation of IVN simulation, with hardware CAROT ECU.

### B. Attack Scenario

In our experiment, we consider two attack scenarios.

1) The attacker has gained root access to the host OS of the ECU, either by physically connecting to the ECU or through a remote access vulnerability.
2) The attacker has gained complete access to a different ECU on the IVN, and is able to send messages over the CAN bus.

If the attacker has root access to the ECU, a non-TEE-based ECU program can simply be killed, which will cause critical problems in the IVN if the ECU is important to the vehicle's operation. In the latter scenario, the attacker attempts to use a buffer overflow vulnerability in the ECU logic to rewrite the return address and execute arbitrary supplied code, leading to a compromised vehicle. The attacker does not have access to the CAROT ECU, but can inject CAN packets into the IVN, which are then received by the ECU. The buffer overflow vulnerability is caused by a stack buffer that stores the data fields of incoming CAN messages, with faulty bounds checking. Sending repeated CAN packets results in memory past the buffer—in particular, the return address of the function—being overwritten.

In order to manipulate the ECU, the code injection attack is implemented as follows. The attacker sends repeated standard CAN frames, where the data field in each CAN frame contains a portion of a binary shellcode sequence that performs an exit syscall. In RISC-V, this corresponds to the instructions `li a0,1` and `jal 0x10460`. The attacker constructs a `nop` sled by padding these two instructions with leading `nop` instructions, creating many memory addresses that all lead to the injected code. Finally, the attacker appends the address of the buffer to the end of the sequence. The choice

of address is a well-informed guess, since the `nop` sled will handle any address that is arbitrarily close. The final effect is a byte sequence that contains the `nop` sled, injection code, and the address to overwrite the return address with. This byte sequence is split across multiple CAN packets and sent to the ECU, triggering the buffer overflow vulnerability and overwriting the return address to somewhere in the `nop` sled, thus successfully performing a code injection attack.

### C. Performance Results

We report performance benchmarks for both the fully implemented CAROT ECU and the baseline. We measure minimum, maximum, and average latency in calculating steering angle given lane lines, measured across a 5-minute simulation. The measurements do not include the CV pipeline, but include CAN bus overhead, the calculation time for the ECU, and OS overhead, if applicable. The results are shown in Table III.

TABLE III
ECU LATENCY. COMPARISON OF BASELINE IMPLEMENTATION WITH FULL IMPLEMENTATION CONTAINING ISR AND KEYSTONE.

| | Baseline | Full | Increase |
|---|---|---|---|
| Min | 120.14 ms | 119.58 ms | -0.46% |
| Max | 139.36 ms | 141.40 ms | 1.46% |
| Average | 129.18 ms | 130.54 ms | 1.05% |

In the average case, the full CAROT ECU implementation is only 1.05% (1360 µs) slower than the baseline implementation. The latency range is also quite small, so this average slowdown is quite predictable. TEECheck, the project closest in scope to ours, incurs an additional 477 µs latency in contrast to their baseline [19]. Hence, CAROT adds 883 µs of message-sending overhead in comparison to TEECheck. Considering a 100+ MHz implementation, this translates to 680 µs additional latency (441 µs when compared to TEECheck). According to real-world benchmarks, both these relative and absolute latency increases are quite reasonable and do not violate timing constraints, as most ECUs send messages in intervals of tens of milliseconds [37].

### D. Security Results

We give the results of the attack scenarios, performed on both the base and full ECUs. With root access to the host OS, the attacker is easily able to kill the running process. The attacker can also write their own executables to replace the original ECU logic, completely compromising the operation of the vehicle. On the other hand, in the full CAROT ECU implementation, the attacker gaining root access does not affect the operation of the vehicle. The ECU logic is completely isolated from the host OS through the separation kernel, which the attacker still does not have access to. Hence, the CAROT ECU defends against a compromised OS scenario.

With access to another ECU on the IVN, the attacker is able to send and receive CAN packets on the network. Using a carefully crafted message sequence, the attacker is able to run arbitrary shellcode on the ECU—specifically, an exit sequence that terminates operation of the ECU. Since the ECU is critical to vehicle operation, this causes a systemwide failure. In contrast, when the attacker attempts to perform the same code injection attack on the full CAROT ECU, the inserted shellcode is interpreted as meaningless instructions by the CPU, due to the ISR mechanism. Hence, an illegal instruction exception is triggered, which is caught by the runtime. The runtime then jumps to the predefined fail-safe controller, preprogrammed to bring the car to a halt.

## VII. RELATED WORK

CAN bus security techniques have been an active subject of research, broadly divided into physical layer and network layer methods. Intrusion Detection Systems (IDS) have been developed for both the physical and network layers. IDS mechanisms monitor the CAN bus, watching for any unusual events which may indicate an intrusion and raising an alarm if such events are detected. These intrusions may be internal or external, as long as their behavior is reflected in CAN bus packets. IDS are either passive, if the system only detects intrusions; or active, if the system acts to prevent intrusions as well [38]. To detect intrusions, IDS either use physical layer phenomena, such as voltage fluctuations and timing deviations; or network-layer events, such as unexpected identifiers and frequency anomalies. Across both layers, researchers have developed general algorithms [6], [7], as well as machine learning-based approaches [8]–[10] for intrusion detection.

However, IDS methods are still generally responsive in nature: they require a dedicated bus monitoring node and only detect an attacker once they have gained access to the CAN bus. Additionally, there is the possibility of the attacker being able to learn CAN message patterns or IDS weaknesses and circumvent the IDS defenses. In comparison, our method completely contains an attacker within the compromised node and prevents them from accessing the CAN bus at all. Further, even if an attacker manages to gain access to the CAN bus, our approach protects other CAROT ECUs on the bus against compromising injection attacks.

Researchers have also developed authentication-based schemes in the network layer. For the CAN 2.0B protocol, CANAuth [11], vatiCAN [12], LeiA [13], LiBrA-CAN [14], and CaCAN [15] all provide solutions that use some variation of Message Authentication Codes (MACs). These methods incur a large overhead on the receiver side, which can also be unnecessary since unauthorized packets are still transmitted before they are authenticated. In addition, authentication approaches do not provide end-to-end protection, and instead only secure the CAN bus interface. An attacker who gains access to an ECU will still be able to manipulate the features managed by that ECU.

Trust has also been introduced as a method to secure ECUs and the CAN bus, but still usually in the context of message authentication. VeCure implements a trust group structure in the IVN, where a “high-trust” group controls critical ECU infrastructure and a “low-trust” group manages

telematics and other vulnerable surfaces [16]. Gui et al. propose a system based on a hardware Trusted Platform Module (TPM), which provides secure storage and cryptographic key generation for ECU logic [17]. The VulCAN platform expands on LeiA and vatiCAN by providing a TCB, inside which the MAC generation process is implemented [18]. While none of these platforms explicitly use TEEs, the TEECheck project—arguably the closest in scope to ours—uses the ARM TrustZone TEE platform in its implementation. TEECheck provides message authentication within a TEE to prevent masquerade and Denial-of-Service (DoS) attacks on the CAN bus [19]. In contrast to these trust-based and regular authentication schemes, our platform protects the CAN bus from unauthorized access. Moreover, our system also addresses the issue of vulnerabilities within the ECU logic code.

## VIII. Conclusion

In this paper, we have designed a secure and efficient open-source automotive ECU using RISC-V and TEEs. We incorporated the ISR MTD technique to counter code injection attacks, a significant threat to ECUs in practice. We have implemented a prototype of our proposed platform on real RISC-V hardware, providing isolation of ECU logic inside of a TEE, guaranteed ISA privacy using ISR, and secure CAN bus access using custom drivers for the TEE platform of choice. We have further evaluated the platform in a realistic setting, demonstrating that the ECU is secure against common code injection and local OS attacks, while maintaining a small performance overhead. Furthermore, in the event of a triggered error state, we establish full control of the system using a fail-safe controller.

In its current implementation, CAROT has several limitations. Due to the nature of the TEE implementation, CAROT does not guarantee security against side-channel attacks, physical bus attacks, and other vulnerabilities inherent to the hardware. Moreover, the current XOR-based ISR implementation is lightweight but fails to protect against more sophisticated attackers. Finally, CAROT faces real-world adoption challenges, including RISC-V's PMP region limit (16 by default, with an optional extension supporting up to 64), which limits the maximum amount of TEEs and peripherals that can be supported. Complex ECUs that require more compute power, communication, and sensor input are thus difficult to implement using CAROT.

We plan to address these limitations in future work. First, we will enhance security of the I/O path through memory encryption and cache partitioning, preventing a host of hardware-level vulnerabilities like Rowhammer [39] and cold-boot attacks [40]. Second, we plan to evolve ISR into the continuous runtime code randomization MTD, which will provide protection against sophisticated exploits like return-oriented programming [41]. Lastly, we aim to evaluate CAROT under more realistic scenarios and complicated ECU implementations, particularly in multi-ECU settings with complex control algorithms (e.g., model-predictive controllers) beyond proportional control.

## Acknowledgments

This work is based upon the work supported by the National Center for Transportation Cybersecurity and Resiliency (TraCR) (a U.S. Department of Transportation National University Transportation Center) headquartered at Clemson University, Clemson, South Carolina, USA. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of TraCR, and the U.S. Government assumes no liability for the contents or use thereof.

## References

[1] W. Zeng, M. A. S. Khalid, and S. Chowdhury, "In-Vehicle Networks Outlook: Achievements and Challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1552–1571, 2016, conference Name: IEEE Communications Surveys & Tutorials. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7393668

[2] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaâniche, and Y. Laarouchi, "Survey on security threats and protection mechanisms in embedded automotive networks," in *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, Jun. 2013, pp. 1–12, iSSN: 2325-6664. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6615528?casa_token=6JKTChiyLuQAAAAA:0hc7qum4h2-3aj6I407MTyTAthyWx8Z8asBxvLzDou--_bjcrv67FcILRrCiQnkPG9C-suAU

[3] C. Valasek and C. Miller, "Adventures in Automotive Networks and Control Units," Seattle, WA, Tech. Rep., 2014. [Online]. Available: https://ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf

[4] ——, "A Survey of Remote Automotive Attack Surfaces," Seattle, WA, Technical White Paper, Jul. 2014. [Online]. Available: https://ioactive.com/pdfs/IOActive_Remote_Attack_Surfaces.pdf

[5] P. Kleberger, T. Olovsson, and E. Jonsson, "Security aspects of the in-vehicle network in the connected car," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2011, pp. 528–533, iSSN: 1931-0587. [Online]. Available: https://ieeexplore.ieee.org/document/5940525

[6] M. Gmiden, M. H. Gmiden, and H. Trabelsi, "An intrusion detection method for securing in-vehicle CAN bus," in *2016 17th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, Dec. 2016, pp. 176–180. [Online]. Available: https://ieeexplore.ieee.org/document/7952095

[7] H. M. Song, H. R. Kim, and H. K. Kim, "Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network," in *2016 International Conference on Information Networking (ICOIN)*, Jan. 2016, pp. 63–68. [Online]. Available: https://ieeexplore.ieee.org/document/7427089

[8] M. Hanselmann, T. Strauss, K. Dormann, and H. Ulmer, "CANet: An Unsupervised Intrusion Detection System for High Dimensional CAN Bus Data," *IEEE Access*, vol. 8, pp. 58 194–58 205, 2020, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/document/9044377

[9] M. D. Hossain, H. Inoue, H. Ochiai, D. Fall, and Y. Kadobayashi, "LSTM-Based Intrusion Detection System for In-Vehicle Can Bus Communications," *IEEE Access*, vol. 8, pp. 185 489–185 502, 2020, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/document/9216166

[10] M. R. Moore, R. A. Bridges, F. L. Combs, M. S. Starr, and S. J. Prowell, "Modeling inter-signal arrival times for accurate detection of CAN bus signal injection attacks: a data-driven approach to in-vehicle intrusion detection," in *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*. Oak Ridge Tennessee USA: ACM, Apr. 2017, pp. 1–4. [Online]. Available: https://dl.acm.org/doi/10.1145/3064814.3064816

[11] A. Herrewege, D. Singelée, and I. Verbauwhede, "CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus," Jan. 2011, p. 7.

[12] S. Nürnberger and C. Rossow, "– vatiCAN – Vetted, Authenticated CAN Bus," in *Cryptographic Hardware and Embedded Systems – CHES 2016*, B. Gierlichs and A. Y. Poschmann, Eds. Berlin, Heidelberg: Springer, 2016, pp. 106–124.

[13] A. Radu and F. Garcia, "Leia: A lightweight authentication protocol for can," vol. 9879, 09 2016, pp. 283–300.

[14] B. Groza, S. Murvay, A. van Herrewege, and I. Verbauwhede, "LiBrA-CAN: A Lightweight Broadcast Authentication Protocol for Controller Area Networks," in *Cryptology and Network Security*, J. Pieprzyk, A.-R. Sadeghi, and M. Manulis, Eds. Berlin, Heidelberg: Springer, 2012, pp. 185–200.

[15] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata, "Cacan - centralized authentication system in can," 11 2014.

[16] Q. Wang and S. Sawhney, "VeCure: A practical security framework to protect the CAN bus of vehicles," in *2014 International Conference on the Internet of Things (IOT)*, Oct. 2014, pp. 13–18. [Online]. Available: https://ieeexplore.ieee.org/document/7030108

[17] Y. Gui, A. S. Siddiqui, and F. Saqib, "Hardware Based Root of Trust for Electronic Control Units," in *SoutheastCon 2018*, Apr. 2018, pp. 1–7, iSSN: 1558-058X. [Online]. Available: https://ieeexplore.ieee.org/document/8479266

[18] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC '17. New York, NY, USA: Association for Computing Machinery, Dec. 2017, pp. 225–237. [Online]. Available: https://doi.org/10.1145/3134600.3134623

[19] 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*, ser. RTNS '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 128–138. [Online]. Available: https://doi.org/10.1145/3394810.3394822

[20] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*. USA: Prentice Hall PTR, Jun. 2002.

[21] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64. [Online]. Available: https://ieeexplore.ieee.org/document/7345265

[22] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, 2020.

[23] V. Costan and S. Devadas, "Intel SGX Explained," 2016, publication info: Preprint. [Online]. Available: https://eprint.iacr.org/2016/086

[24] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 130:1–130:36, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3291047

[25] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer Science & Business Media, Aug. 2011, google-Books-ID: VERsnX772S0C.

[26] J.-H. Cho, D. P. Sharma, H. Alavizadeh, S. Yoon, N. Ben-Asher, T. J. Moore, D. S. Kim, H. Lim, and F. F. Nelson, "Toward Proactive, Adaptive Defense: A Survey on Moving Target Defense," *Commun. Surveys Tuts.*, vol. 22, no. 1, pp. 709–745, Jan. 2020. [Online]. Available: https://doi.org/10.1109/COMST.2019.2963791

[27] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 298–307. [Online]. Available: https://doi.org/10.1145/1030083.1030124

[28] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7783743

[29] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/aslrcache-practical-cache-attacks-mmu/

[30] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 191–205, iSSN: 1081-6011. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6547110

[31] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 281–289. [Online]. Available: https://doi.org/10.1145/948109.948147

[32] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 272–280. [Online]. Available: https://doi.org/10.1145/948109.948146

[33] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "Asist: architectural support for instruction set randomization," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 981–992. [Online]. Available: https://doi.org/10.1145/2508859.2516670

[34] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[35] eugene tarassov, "eugene-tarassov/vivado-risc-v," Apr. 2024, original-date: 2020-02-15T01:12:54Z. [Online]. Available: https://github.com/eugene-tarassov/vivado-risc-v

[36] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[37] S. Kramer, D. Ziegenbein, and A. Hamann, "Real World Automotive Benchmarks For Free," 2015.

[38] S.-F. Lokman, A. T. Othman, and M.-H. Abu-Bakar, "Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 184, Jul. 2019. [Online]. Available: https://doi.org/10.1186/s13638-019-1484-3

[39] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 361–372, Jun. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2678373.2665726

[40] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1506409.1506429

[41] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code {Re-Randomization}," 2016, pp. 367–382. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king