carroot: A Secure Automotive ECU for Connected Vehicles

by
Ahmer Raza

Under the Supervision of
Dr. Zhenkai Zhang

In partial fulfillment of the requirements for
Degree of Bachelor of Sciences
with Departmental Honors
in Computer Science

Clemson University
Clemson, SC
May 3, 2024

To Baba, my mentor; Mama, my favorite; and Zuha, my little shadow.

Acknowledgments

My research at Clemson would not have been possible without the following people. I sincerely thank them for their contribution to my academics, curiosity, and intellectual growth.

Dr. Zhenkai Zhang, my research advisor, who has patiently supervised my journey through the realm of cybersecurity. Thank you for the fantastic learning experiences your classes were, thank you for introducing me to research, thank you for our engaging discussions, and thank you for answering my many, many questions!

Dr. Nantsoina C. Ramiharimanana, whose engaging after-class discussions kept me perpetually curious. Thank you for your sincere advice and guidance in navigating academia!

Dr. William J. Reid III, whose Robotics CI gave me my first hands-on project experience at Clemson. Thank you for your infinite encouragement, and thank you for the brownie!

Finally, Dr. Hassan Raza. I cannot thank you enough for all you have done for me, at Clemson and elsewhere. I thank you for your constant mentorship and thoughtful decisions, which have always benefitted me, even if I was too shortsighted to see it. I thank you for the lifetime of intellectual development you have provided that has allowed me to flourish academically. I thank you for everything, from Whitey's Ice Cream to Lake View Park. Thank you!

Contents

List of Figures

Code Listings

Chapter 1

Introduction

Cars are no longer simple mechanical devices. The average car contains a complex network of Electronic Control Units (ECUs), each used to control specific components of the car. There are ECUs for powertrain management, engine management, infotainment, intruder detection, safety systems, telematics, etc. [1]. All of these ECUs are linked together through an In-Vehicle Network (IVN), allowing for communication and other data transfer throughout the vehicle.

1.1 Motivation

Researchers have shed light on the fact that ECUs and IVNs have not been developed from a security-first perspective. The ECUs themselves do not provide the basic requirements of confidentiality, authenticity, availability, integrity, or non-repudiation [2]. As a result, many fixed computer vulnerabilities have been found to affect ECUs, and many new ECU-specific vulnerabilities have also been found [2–5]. As vehicles are a critical backbone of modern infrastructure, any security vulnerability in their systems would have potentially severe consequences.

The problem of trusted computing has also been a persistent one. After much evaluation of the conventional Rich Execution Environment (REE), in which an untrusted host operating system executes untrusted code, a platform called the Trusted Execution Environment (TEE) has been devised. A TEE provides an isolated environment for executing trusted code, incorporating many features such as memory protection, trust measurement, trusted I/O and memory, root-of-trust, etc. We propose that a secure automotive ECU (and, thus, IVN) can be constructed by incorporating TEEs into existing designs for ECUs.

1.2 Thesis Overview

We first introduce the current design, organization, and security implications of ECUs and IVNs in Chapter 2. Mainly, we focus on the bus technologies used in IVNs, as this is where most of the exploitation mechanisms lie. We then introduce the concept of TEEs in Chapter 3, including REE vs. TEE, fundamental trust concepts, and the Keystone project. Finally, in Chapter 4, we propose a solution to the insecure ECU problem: carroot. We give a theoretical platform for the ECU, evaluate whether it safeguards against threats to the ECU, and detail our physical implementation of this ECU. We also describe our experience porting Keystone to a new, unsupported platform.

Chapter 2

Connected Automotive Control

This chapter describes the organization of controllers inside modern cars, namely, how Electronic Control Units (ECUs) are arranged in In-Vehicle Networks (IVNs), interconnected using buses like CAN, LIN, and MOST.

2.1 ECUs

The automotive ECU forms the backbone of control in modern cars. A typical modern car will use dozens of ECUs to control many simple components, such as lights, wipers, doors, windows, and entertainment systems [6]. Multiple linked ECUs are also used to control more complex systems, such as the Antilock Braking System (ABS), Electronic Stability Program (ESP), Electric Power Steering (EPS), and active suspension mechanisms. ECUs can be implemented in a variety of ways. Microcontroller Units (MCUs), Field Programmable Gate Arrays (FPGAs), and Application-Specific Integrated Circuits (ASICs) are some of the most common.

2.2 In-Vehicle Networks

ECUs within a car communicate with each other via an IVN. The IVN is a multiplexed medium, connecting ECUs without direct connections between them. Prior to the development of this network, ECUs would be connected point-to-point, causing the number of necessary connections to grow quadratically [6]. Linking ECUs to a universal bus allows for linear scaling and less connection redundancy. Figure 2.1 demonstrates the organization of ECUs in a typical vehicle's IVN.

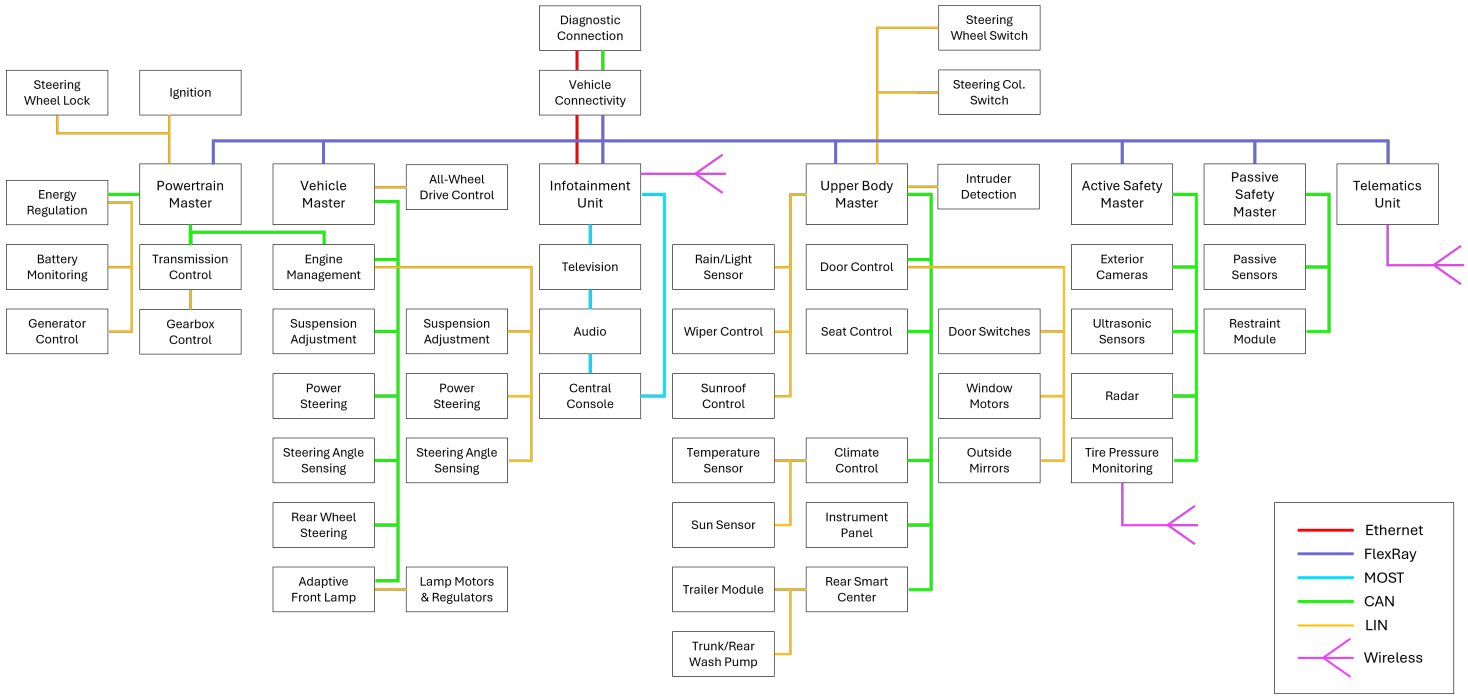

Figure 2.1: Topology of an advanced IVN. Adapted from [1].

2.2.1 Buses

There are four important bus protocols used to implement the multiplexed medium within the IVN: Controller Area Network (CAN), Local Interconnect Network (LIN), Media Oriented Systems Transport (MOST), and FlexRay.

CAN is an ISO-standard message broadcast protocol, and is the most widely used bus in IVNs [7]. A CAN message can be of four different types, or frames: data frame, to send data; remote frame, to request data; error frame, to represent an invalid message; or overload frame, to delay messages for a busy node. Messages transmitted using CAN contain, among other metadata, an 11-bit identifier that stores the priority of the message and the 64-bit message itself [8]. Figure 2.2 shows the format of a CAN message. Nodes in the IVN connect to a CAN bus using a CAN transceiver, which is controlled using a CAN controller. Figure 2.3 shows an example CAN bus with n connected nodes.

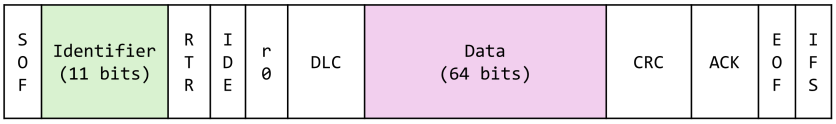

Figure 2.2: The CAN message format. Adapted from [8].

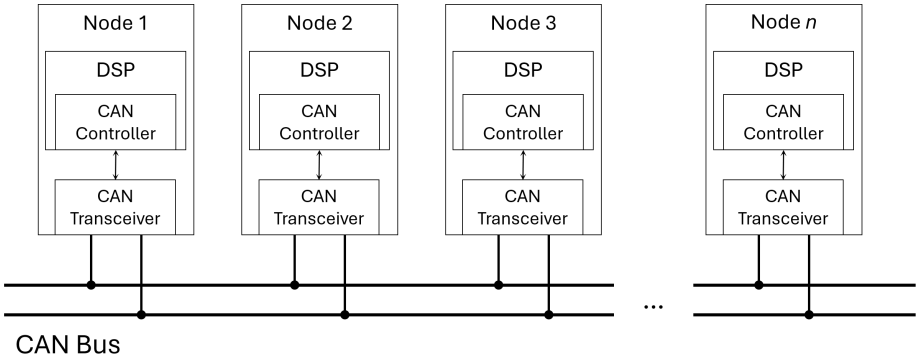

Figure 2.3: An example of n Digital Signal Processing (DSP) nodes in a CAN bus interconnect. Adapted from [8].

LIN is an inexpensive message broadcast protocol that is also used in IVNs. A LIN message simply consists of a frame header and frame response. A LIN bus requires one node to be assigned as commander, and supports up to 16 nodes as responders. The commander node controls all communication on the bus; responder nodes cannot communicate with each other and instead respond to the commander's requests [9].

MOST is a high-speed message broadcast protocol, used exclusively for multimedia transport in an IVN. Nodes in a MOST network must be interconnected in a ring: each node must have exactly 2 neighbors. A MOST message is transmitted through all nodes between the sender and intended receiver. The MOST physical layer uses optical fiber or electrical/coaxial cables, along with the appropriate transceivers, and supports bit rates of 25, 50, and 150 Mbit/s, depending on the hardware [10].

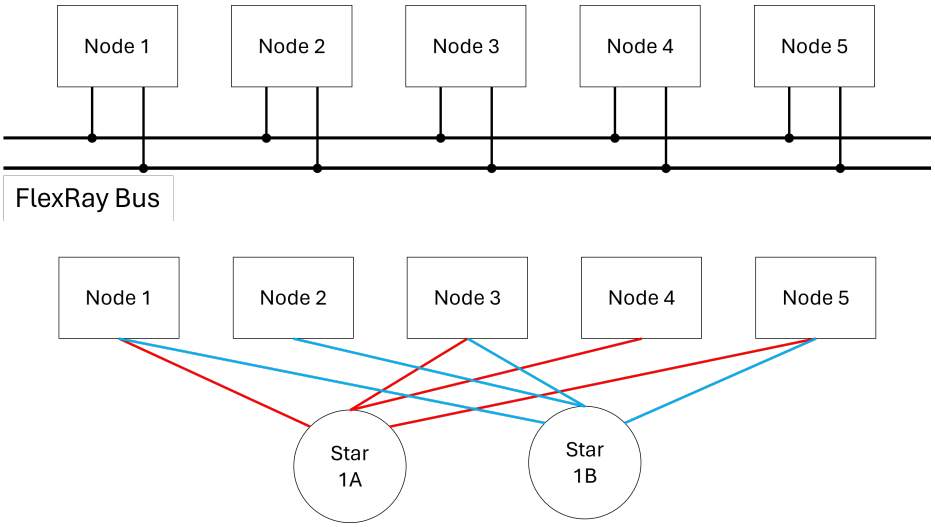

Figure 2.4: Two of the topologies supported by FlexRay: dual-channel bus (above) and dual-channel single-star (below) configurations. Adapted from [11].

FlexRay is a high-speed, fault-tolerant message broadcast protocol that is used in IVNs to connect the most important master control ECUs together. In operation, it is similar to CAN: frames are sent on the bus to a specific node. However, it is versatile regarding topological configuration. A FlexRay

bus can be set up like CAN, with nodes attached to a dual-channel bus. A bus can also be used in a single- or dual-channel star configuration, with two star nodes handling message transmission throughout the network. Finally, a FlexRay bus can also be set up using a combination of the two [11]. Figure 2.4 shows both a passive-bus and an active-star topology.

2.3 Security

The security of IVNs remains an issue. The ECUs themselves have not been designed to be resistant to attacks, and are instead purely focused on functionality. If a single ECU is gained control of, the entire IVN can be manipulated. It has been proven that ECUs do not support the cyber-security principles of confidentiality, authenticity, availability, integrity, and non-repudiation [2]. Thus, many conventional hardware and software attacks are possible on ECUs, some proven examples being replay attacks and code injection. The inclusion of communication buses and wireless transmission in the IVN adds even more vulnerability and opens the door for many new attacks on IVNs. Specifically, remote attacks via wireless transmission and injection attacks via the CAN bus have been demonstrated to be simple for a malicious actor to perform, consequently gaining complete control over the vehicle [2–4].

Chapter 3

Trusted Computing

This chapter compares Rich Execution Environments (REEs) and Trusted Execution Environments (TEEs) and describes related concepts, including trust, trust measurement, and Root-of-Trust (RoT). It then introduces the Keystone Enclave project.

3.1 TEEs

The problem of secure computation is a difficult one. Countless tools have been developed to counteract computational exploits, and countless exploits have been found to nullify these tools, in an endless cycle. Thus far, the conventional computing platform has been the REE, where an untrusted OS kernel runs untrusted code [12]. TEEs are a class of platforms that attempt to mitigate the issues REEs face by executing applications within trusted isolated processing environments, thus guaranteeing the confidentiality of code and data, the integrity of runtime states, and the authenticity of executed code [13].

There are a few fundamental building blocks that a TEE requires. Firstly, there must be a RoT with a secure boot procedure that assures only untampered, validated code is loaded and bootstrapped. There must also be a secure scheduling module that guarantees balanced and efficient scheduling between the REE and the TEEs. Importantly, there should also be an inter-environment communication module, to allow a TEE to communicate with the REE and vice-versa. This communication module must be reliable, efficient, and protected, as this interface with the REE introduces many new threats to the security of the TEE. Additionally, a TEE needs secure storage that provides data confidentiality, integrity, freshness, and authorized access. Finally, there should be a trusted I/O path that guarantees genuineness of data [13]. Figure 3.1 visualizes the layout of these modules.

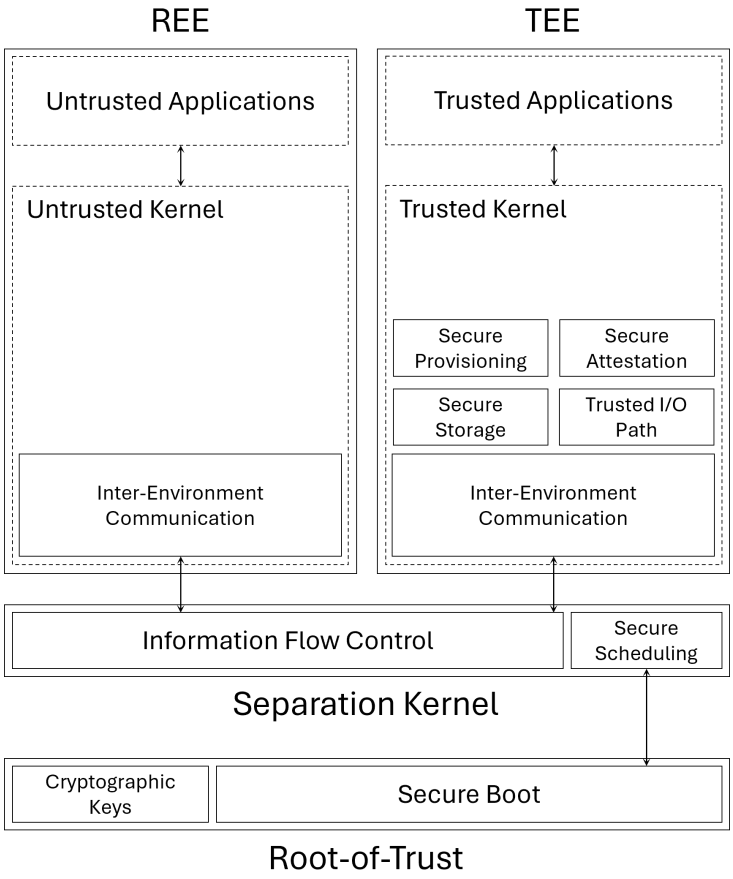

Figure 3.1: The TEE platform. Adapted from [13, 14].

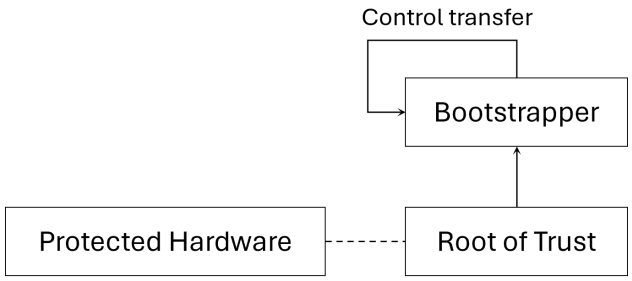

Figure 3.2: A general representation of the secure boot process. Adapted from [15].

3.2 Trust

Trust is fundamental to the concept of the TEE. As discussed, there must be a distinction between trusted and untrusted applications, data, hardware, etc.

3.2.1 Trust Measurement

There are two types of trust: static trust, where trust of a system is measured using some predefined rules, and dynamic trust, where trust is measured using the current state of the system, and so updates throughout the TEE's lifetime [13]. While trust is a qualitative, subjective metric, in computing, trust is centered around the expected behavior of a system; if a system behaves reliably and as expected, it is trustworthy.

3.2.2 Root-of-Trust

The RoT is one of the fundamental prerequisites for a TEE. The role of the RoT is twofold: it must give a reliable measurement of the current trust of the system, either static or dynamic, and it must compute a trust score based on the trust measurement performed prior [13]. The trust function used to compute the trust score varies based on requirements, but generally, it considers the level of protection certification, the reliability of the RoT, and the measured integrity of the system. The RoT interacts closely with protected hardware, and performs measurement of the bootstrapping (and related) firmware. Figure 3.2 describes this process.

3.3 Keystone

Most major vendors have provided physical implementations of TEEs: Intel's SGX platform, ARM's TrustZone platform, and AMD's SEV platform, to name a few. However, each of these TEE platforms implements its own set of features, leading to platform-specific nuances and limits. Furthermore, these platforms are all proprietary, meaning there is no room for independent development or understanding of each platform's inner workings. Users of TEEs are thus stuck using a small subset of all the possible features of an

ideal TEE, forced to wait for support for their desired features [16].

Keystone was developed as an open-source, modular, highly customizable TEE platform for RISC-V. The platform consists of a machine-mode Security Monitor (SM) Supervisor Binary Interface (SBI) firmware; a supervisor-mode application runtime (RT); and a user-mode application development library (SDK). Keystone provides enclaves as an isolated computing primitive, which are independent execution environments that contain an instance of the RT and an executable enclave app (eapp) developed using the SDK. The untrusted OS and the RT both call the SBI functions provided by the SM, including enclave create, run, destroy, etc. Keystone natively trusts RISC-V protected hardware without modification, but requires a small secure-boot procedure to be implemented on top of an existing boot ROM or ZSBL (Zero-Stage Bootloader). Native RISC-V Physical Memory Protection (PMP) is used to provide secure storage to applications executed in an enclave [17]. Figure 3.3 shows the organization of Keystone components.

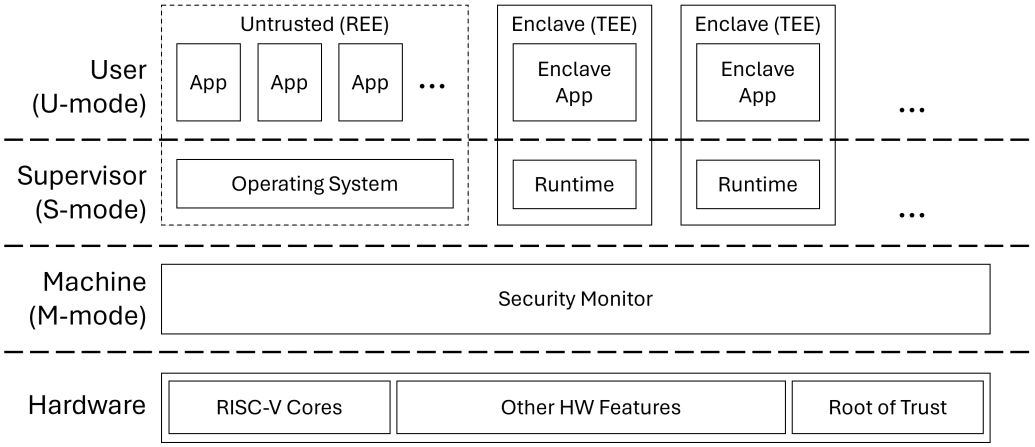

Figure 3.3: Keystone components and platform. Adapted from [17].

Chapter 4

carroot

This chapter describes a proposed solution to insecure IVNs: carroot. It describes how TEEs can be used to implement a secure automotive ECU. Additionally, it details current development towards a physical implementation of this platform.

4.1 Overview

As discussed in Chapter 2, automotive IVNs are conventionally unprotected and face many threats. As a solution to this problem, we propose a complete, secure implementation of ECUs that provide end-to-end security within IVNs. By sandboxing the execution of code within ECUs and providing a common RoT, we ensure defense against many typical attacks to which IVNs are susceptible. We support communication between ECUs using the CAN protocol, as it is the most prevalent bus used in IVNs.

4.2 Proposed Platform

A carroot-based IVN consists of multiple carroot ECUs interconnected via a CAN bus. As no modification to the bus is required, each carroot ECU is, for the most part, integratable without much modification to the underlying infrastructure. A carroot ECU consists of a RISC-V System-on-Chip (SoC) that provides a RoT. The SoC runs a Linux-based host OS and a TEE platform used to execute trusted code relevant to the role of the specific ECU. The ECU logic partially runs on the host OS, but mostly is encapsulated and runs in a TEE. Figure 4.1 visualizes the components of a carroot ECU attached to a CAN bus.

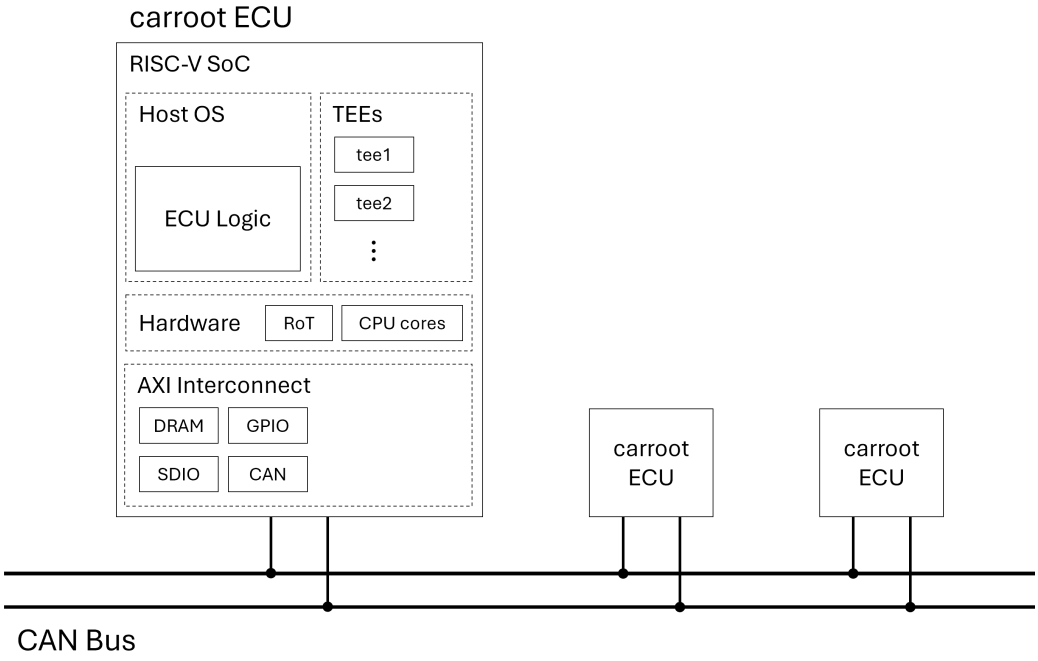

Figure 4.1: The carroot ECU and an example carroot platform with 3 ECUs attached to a CAN bus.

4.2.1 Threat Model

As discussed, injection attacks via the CAN bus and remote attacks via the wireless transceiver pose the biggest threat to the IVN [18]. Hence, we consider these attacks as the foremost threats in our threat model. Given that a root-of-trust is provided, a TEE safeguards against all these attacks. For two specific examples: code injection involves the execution of untrusted code, which is prevented by the TEE itself. Also, because the TEE provides protected I/O with guaranteed memory freshness, replay attacks are averted too.

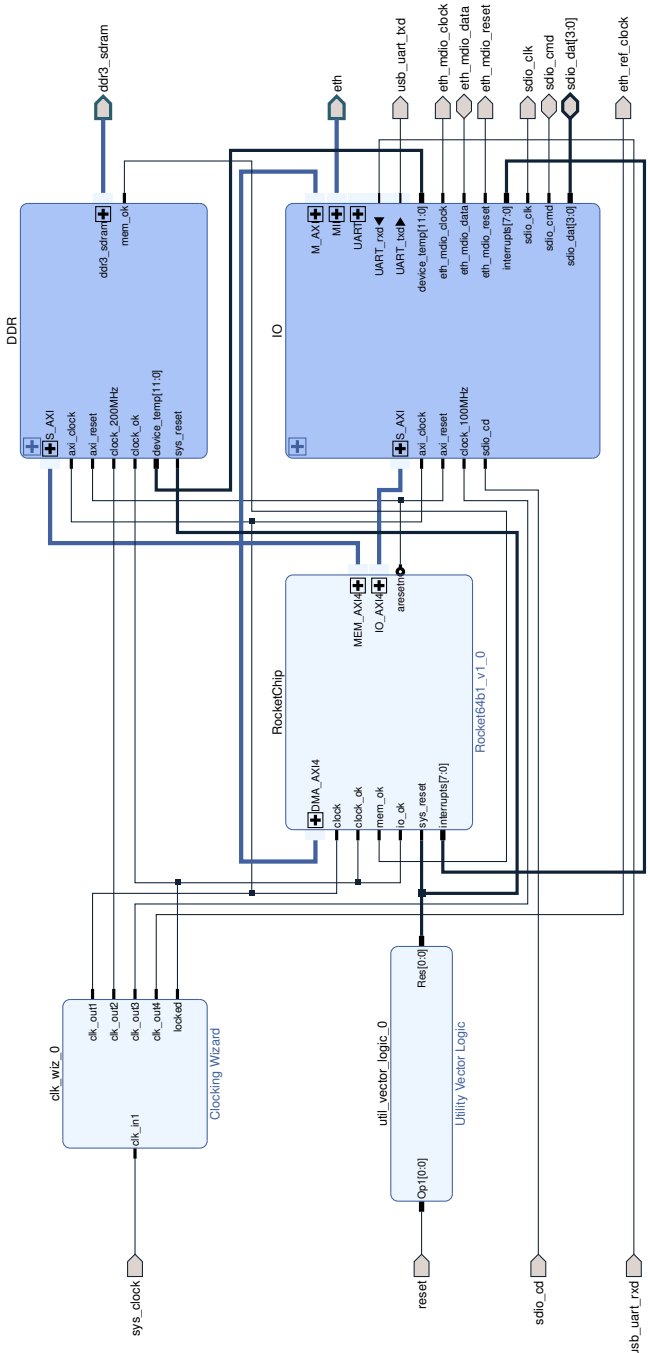

Figure 4.2: Our current implementation of the carroot SoC.

4.3 Physical Implementation

We partially prototype a physical implementation of the carroot ECU. Our SoC consists of a single Rocket RISC-V softcore implemented on an Arty A7-100T FPGA board. DRAM, UART, SDIO, etc. are connected to the CPU via AXI buses on the board. The block designs for this SoC are provided by Eugene Tarassov [19]. Figure 4.2 shows our used block design.

We also integrate the Keystone software stack on this SoC. We modify the Rocket core boot ROM to become a root-of-trust and provide the secure boot procedure to Keystone firmware, including cryptographic key generation and SM measurement. We also integrate the SM and related firmware with the SBI. Finally, we boot a Debian Linux image on this platform, with ongoing work to integrate the Keystone RT and SDK.

4.4 Porting Keystone

As Keystone is a modular platform, it is simple to integrate into an existing, working platform. However, the platforms Keystone provides by default, including OpenSBI, Buildroot, etc., do not natively support Rocket-based systems. To run Keystone on the Arty A7-100T FPGA board, we explore several directions. In our discussion, we assume file and directory paths are relative to the base directory of either Eugene Tarassov's `vivado-risc-v` repo [19] or the Keystone repo [20].

4.4.1 Porting Buildroot and OpenSBI

After deciding on and implementing the Rocket softcore CPU on the Arty board, we try to port Buildroot and OpenSBI to the developed Rocket platform. While Buildroot Linux can be built, adding the proper embedded device tree, drivers, etc. is complex. Furthermore, the OpenSBI configuration Keystone provides is difficult to port to the Rocket system, due in part to the old version Keystone uses (v0.9) in comparison to the one provided by Eugene Tarassov [19] (v1.4). We thus abandon this direction, in the interest of time.

4.4.2 Porting Keystone SM, RT, and SDK

We explore disassembling the Keystone SM, RT, and SDK from its default platform and integrating these components into the working OpenSBI firmware and Debian Linux OS. Thus far, we have successfully implemented the Keystone prerequisites (secure boot, SM measurement, etc.) on our Rocket system, and have also integrated the Keystone SM into our SBI. We are actively working on integrating the RT and SDK as well.

The first step is to provide a secure in the ZSBL (Zero Stage Bootloader, also known as boot ROM) for the Keystone SM. Keystone provides a sample bootloader in the `bootrom/bootloader.c` file. We simply incorporate the bootloader function and its dependencies into the `bootrom/bootrom.c` file provided by Eugene Tarassov. The `bootloader()` function is called by the `download()` function, after the OpenSBI firmware is loaded but before it is executed. It could alternatively be called before the OpenSBI firmware is loaded in the first place, by the top-level `main()` function.

Importantly, Keystone requires 352 (0x160) bytes of memory for the ZSBL to store cryptographic keys, hashes, and signatures. As DRAM is entirely self-managed at boot time, this data must be placed at a very calculated location, else it may be overwritten or overwrite other important data. Using Eugene Tarassov's platform, we place this sequence starting at DRAM address 0x88000000, which is unused by every other boot process. Listing 1 gives the definition placed in the `bootrom/bootrom.c` file to incorporate the necessary keys for the ZSBL. The bootrom cannot contain any memory-loaded `.data` or `.bss` sections, so the data locations cannot be included in a linker script and must be accessed directly by the ZSBL.

```
// bootrom/bootrom.c

static byte *sanctum_dev_public_key  = (byte *)0x88000020;
static byte *sanctum_dev_secret_key  = (byte *)0x88000040;
static byte *sanctum_sm_hash         = (byte *)0x88000080;
static byte *sanctum_sm_public_key   = (byte *)0x880000C0;
static byte *sanctum_sm_secret_key   = (byte *)0x880000E0;
static byte *sanctum_sm_signature    = (byte *)0x88000120;
```

Listing 1: The memory addresses where the required data for the Keystone SM was placed.

We also incorporate the Keystone SM itself into the OpenSBI firmware. This requires more modification, as Keystone uses a different OpenSBI version than what is provided by Eugene Tarassov—v0.9 and v1.4, respectively. We first include the Keystone SM as an OpenSBI library. We split the `sm/src` folder into two, both with the same directory tree: one for `.h` files and one for `.c` files. We then place the former in the `opensbi/include/sbi_utils` directory, and the latter in the `opensbi/lib/utils` directory. Next, we modifiy the `platform.c` file to provide the `sm_init()` function as the OpenSBI final init procedure. Listing 2 shows this.

```
// opensbi/platform/vivado-risc-v/platform.c

static int rocket_final_init(bool cold_boot)
{
    sm_init(cold_boot);
    return 0;
}

const struct sbi_platform_operations platform_ops = {
    ...
    .timer_exit = fdt_timer_exit,
    .final_init = rocket_final_init, // added
};
```

Listing 2: The modified OpenSBI platform specifier.

As Keystone uses a different version of OpenSBI, there is one significant change to incorporate. The Translation Lookaside Buffer (TLB), which Keystone manages, is handled differently in OpenSBI v1.4 than v0.9. Specifically, in v0.9, the `struct sbi_tlb_info` used to specify TLB settings accepts a local update function, whereas v1.4 accepts an `enum sbi_tlb_type` that corresponds to a `static` function. To incorporate Keystone's custom TLB local update function, we make modifications in various files. Firstly, we add an entry to `enum sbi_tlb_type` in file `opensbi/include/sbi/sbi_tlb.h` corresponding to an option for the custom Keystone local update function. Secondly, we add a `case` statement in the `tlb_entry_local_process()` function in file `opensbi/lib/sbi/sbi_tlb.c` that calls the `sbi_pmp_ipi_local_update()` function (including the `sbi_utils/sm/ipi.h` header file prior as well). Thirdly, we create two PMU events in `enum sbi_pmu_fw_event_code_id`. Finally, we add an entry to `u32 tlb_type_to_pmu_fw_event[]` to convert a TLB type to the new PMU event. Listings 3, 4, and 5 show the exact files and code changes.

```
// opensbi/include/sbi/sbi_tlb.h

enum sbi_tlb_type {
    ...
    SBI_TLB_HFENCE_VVMA,
    SBI_TLB_KEYSTONE, // added
    SBI_TLB_TYPE_MAX,
};
```

Listing 3: The modified OpenSBI TLB type struct.

```
// opensbi/lib/sbi/sbi_tlb.c

...
#include <sbi_utils/sm/ipi.h> // added
...
static void tlb_entry_local_process(struct sbi_tlb_info *data)
{
    ...
    case SBI_TLB_KEYSTONE: // added
        sbi_pmp_ipi_local_update(data);
        break;
    ...
}
...
static const u32 tlb_type_to_pmu_fw_event[SBI_TLB_TYPE_MAX] = {
    ...
    [SBI_TLB_HFENCE_VVMA] = SBI_PMU_FW_HFENCE_VVMA_SENT,
    [SBI_TLB_KEYSTONE] = SBI_PMU_KEYSTONE_SENT, // added
};
...
```

Listing 4: The modified OpenSBI local function caller.

```
// opensbi/include/sbi/sbi_ecall_interface.h

enum sbi_pmu_fw_event_code_id {
    ...
    SBI_PMU_KEYSTONE_SENT = 22, // added
    SBI_PMU_KEYSTONE_RCVD = 23, // added

    SBI_PMU_FW_MAX,
    ...
};
```

Listing 5: The modified OpenSBI firmware event types.

We also have to slightly modify Keystone's `sm_copy_key()` function to copy the keys from the correct address. We copy the `sm/platform/cva6` folder into a `vivado-risc-v` folder in the same directory, and modify the newly-created `vivado-risc-v/platform.c` to perform the correct key copy procedure from the correct memory location. Listing 6 shows the exact changes.

```
// opensbi/lib/utils/sm/platform/vivado-risc-v/platform.c

// all added or modified

static byte *sanctum_dev_public_key  = (byte *)0x88000020;
static byte *sanctum_sm_hash         = (byte *)0x88000080;
static byte *sanctum_sm_public_key   = (byte *)0x880000C0;
static byte *sanctum_sm_secret_key   = (byte *)0x880000E0;
static byte *sanctum_sm_signature    = (byte *)0x88000120;

extern byte sm_hash[MDSIZE];
extern byte sm_signature[SIGNATURE_SIZE];
extern byte sm_public_key[PUBLIC_KEY_SIZE];
extern byte sm_private_key[PRIVATE_KEY_SIZE];
extern byte dev_public_key[PUBLIC_KEY_SIZE];

void sm_copy_key(void)
{
  sbi_memcpy(sm_hash, sanctum_sm_hash, MDSIZE);
  sbi_memcpy(sm_signature, sanctum_sm_signature, SIGNATURE_SIZE);
  sbi_memcpy(sm_public_key, sanctum_sm_public_key, PUBLIC_KEY_SIZE);
  sbi_memcpy(sm_private_key, sanctum_sm_secret_key, PRIVATE_KEY_SIZE);
  sbi_memcpy(dev_public_key, sanctum_dev_public_key, PUBLIC_KEY_SIZE);
}
...
```

Listing 6: The modified Keystone key copy procedure and addresses.

Finally, we modify some of the Keystone SM configuration files. Firstly, we add a dummy `Kconfig` to the `opensbi/lib/utils/sm` directory. More importantly, we modify the `objects.mk` file to include Keystone's firmware configuration. Listing 7 shows the exact code additions.

```
# opensbi/lib/utils/sm/objects.mk

# all added

KEYSTONE_SDK=/home/araza/current/keystone/sdk
KEYSTONE_PLATFORM=fpga/rocket
KEYSTONE_SOURCES=$(src_dir)/lib/utils/sm
KEYSTONE_HEADERS=$(src_dir)/include/sbi_utils/sm

# from sm/plat config.mk
platform-cflags-y = -I$(KEYSTONE_HEADERS)\
    \-I$(src_dir)/platform/$(KEYSTONE_PLATFORM)/include\
    \-I$(KEYSTONE_SDK)/include/shared

# from sm/plat objects.mk
platform-genflags-y +=\
    \"-DTARGET_PLATFORM_HEADER=\"platform/\
    \$(KEYSTONE_PLATFORM)/platform.h\""

include £(KEYSTONE_SOURCES)/objects.mk
libsbiutils-objs-y +=\
    \$(addprefix sm/,$(subst .c,.o,$(keystone-sm-sources)))
libsbiutils-objs-y +=\
    \sm/platform/$(KEYSTONE_PLATFORM)/platform.o

carray-platform_override_modules-y += platform
```

Listing 7: The additions to the OpenSBI config.mk file.

Chapter 5

Conclusion

As automotive vehicles continue becoming more advanced, concern for their security will grow proportionally. As current automotive ECUs do not protect the security of the vehicle, a redesign is necessary. In this thesis, we covered a RISC-V-based, TEE-centric ECU design that will protect the availability, reliability, and integrity of automotive ECUs. We analyzed the threat model of current ECUs and guaranteed security against the threats they face. We also partially implemented the ECU, with ongoing work to complete the first physical prototype.

5.1 Future Directions

The carroot ECU is only one implementation of a secure ECU. Many researchers have implemented alternative solutions, using a more bottom-up, complete approach [21–23]. Also, as car control technology inevitably changes and new threats inevitably emerge, there will be more room for research in threat mitigation systems. Finally, the TEE itself is also still in its budding phase. Keystone, one of the only mainstream, "complete" open-source TEEs, is still experimental and under development. Finally, trust and trustworthy code is still being studied. Development of effective, dependable metrics for measuring the trustworthiness of a program would benefit not only the maturation of TEEs, but the field of cybersecurity as a whole.

Bibliography

[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, "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

[4] ——, "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

[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] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, Jun. 2005, conference Name: Proceedings of the IEEE. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1435746

[7] W. Choi, H. J. Jo, S. Woo, J. Y. Chun, J. Park, and D. H. Lee, "Identifying ECUs Using Inimitable Characteristics of Signals in Controller Area Networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 6, pp. 4757–4770, Jun. 2018, conference Name: IEEE Transactions on Vehicular Technology. [Online]. Available: https://ieeexplore.ieee.org/document/8303766

[8] S. Corrigan, "Introduction to the Controller Area Network (CAN)," Dallas, TX, Tech. Rep. SLOA101B, May 2016. [Online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf

[9] E. Hackett, "LIN Protocol and Physical Layer Requirements," Dallas, TX, Tech. Rep. SLLA383A, Feb. 2022. [Online]. Available: https://www.ti.com/lit/an/slla383a/slla383a.pdf

[10] A. Grzemba, *MOST: The Automotive Multimedia Network*, 2nd ed. FRANZIS, Jan. 2011. [Online]. Available: https://www2.ciando.com/img/books/extract/3645250611_lp.pdf

[11] "FlexRay Communications System," Jun. 2004. [Online]. Available: http://www.eskorea.net/html/data/support/FlexRay2.0.pdf

[12] J. Jang, S. Kong, M. Kim, D. Kim, and B. Kang, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment," in *NDSS*, Jan. 2015, pp. 1–15.

[13] 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

[14] J. González, "Operating System Support for Run-Time Security with a Trusted Execution Environment," Ph.D. dissertation, Mar. 2015.

[15] H. Löhr, A.-R. Sadeghi, and M. Winandy, "Patterns for Secure Boot and Secure Storage in Computer Systems," in *2010 International Conference on Availability, Reliability and Security*, Feb. 2010, pp. 569–573. [Online]. Available: https://ieeexplore.ieee.org/document/5438035

[16] D. Lee, "Building Trusted Execution Environments," Ph.D., University of California, Berkeley, United States – California, 2022,

iSBN: 9798351476582. [Online]. Available: https://www.proquest.com/docview/2730786496/abstract/2DFD577A9C8F4005PQ/1

[17] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, 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. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3342195.3387532

[18] D. K. Nilsson, P. H. Phung, and U. E. Larson, "Vehicle ECU classification based on safety-security characteristics," in *IET Road Transport Information and Control - RTIC 2008 and ITS United Kingdom Members' Conference*, May 2008, pp. 1–7, iSSN: 0537-9989. [Online]. Available: https://ieeexplore.ieee.org/document/4562233

[19] 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

[20] "keystone-enclave/keystone," May 2024, original-date: 2018-06-12T20:23:19Z. [Online]. Available: https://github.com/keystone-enclave/keystone

[21] S. Woo, H. J. Jo, I. S. Kim, and D. H. Lee, "A Practical Security Architecture for In-Vehicle CAN-FD," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 8, pp. 2248–2261, Aug. 2016, conference Name: IEEE Transactions on Intelligent Transportation Systems. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7435304?casa_token=vFKD8C7jIh0AAAAA:fKQfD4kBDiu-Ubn4C61h3phEFb1Mrt4QyensTvyx7VILXBmdwhIQhh_OqjbX7dHYE_gi4bfE

[22] B. Poudel and A. Munir, "Design and evaluation of a novel ECU architecture for secure and dependable automotive CPS," in *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Jan. 2017, pp. 841–847, iSSN: 2331-9860. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7983243?casa_token=uokHrcl_U5kAAAAA:sc3I3PTXFupleTpLhz2fyq7Xb_

rXNVY0hJpHTUGeg_1kWrY5hbj09GeCFwGSNRLmy5Sxn1-P&signout=success

[23] L. Yu, J. Deng, R. R. Brooks, and S. B. Yun, "Automobile ECU Design to Avoid Data Tampering," in *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, ser. CISR '15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 1–4. [Online]. Available: https://dl.acm.org/doi/10.1145/2746266.2746276