A MOVING TARGET DEFENSE APPROACH TOWARDS SECURITY AND RESILIENCE IN CYBER-PHYSICAL SYSTEMS

By

Bradley Potteiger

Dissertation
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
in

ELECTRICAL ENGINEERING

October 31, 2019

Nashville, Tennessee

Dedicated to my family.

ACKNOWLEDGMENTS

During my time as a Graduate Student at Vanderbilt I was fortunate to work and collaborate with a number of great people. First of all, I would like to express special thanks to my PhD advisor Dr. Xenofon Koutsoukos for believing in me and providing tremendous mentorship and guidance throughout my research. I would also like to thank the other members of my committee Dr. Janos Sztipanovits, Dr. Gabor Karsai, Dr. Adam Tagert, and Dr. Zhenkai Zhenk for agreeing to serve on my PhD committee, as well as providing tremendous support.

I would like to thank every member of the Institute for Software Integrated Systems for making the last 5 years an unbelievable experience. I would especially like to thank Dr. Abishek Dubey, Dr. Siyuan Dai, Dr. Amin Ghafouri, Dr. William Emfinger, Dr. Himanshu Neema, Peter Volgyesi, Dr. Aron Lazka, Dr. Anastasia Mavridou, Dr. Hamzah Abdel-Aziz, Dr. Goncalo Martins, Dr. Waseem Abbas, Dr. Pranav Kumar, Dr. Raphael Stern, Dr. Carlos Barreto, Dr. Brian Broll, Shreyas Ramakrishna, Dr. Ali Ozdagli, Feiyang Cai, Dimitrios Boursinos, Jiani Li, and Scott Eisele for their technical collaboration throughout my PhD experience. I would also like to express tremendous gratitude to Katie Dey, Frankie King, Regan Williams, Emily Wehby, and Amy Karns for their invaluable logistical and graphic design help. Furthermore, I would like to thank everyone at the Wond'ry for their help throughout my PhD studies including Deanna Meador, Dr. Dave Owens, and Michael Bryant. Finally, I would like to thank Dr. Sandya Lakkur, Jacob Vanderburgh, Ryan Jarret, Tyler Jarret, Shannon Melendy, David Schluter, Jacquelyn Neal, Emma de Caussin, Tim Ault, Tim Krentz, Charlie Hartsell, Candace Carducci, and Ben Yett for their devoted friendship while at Vanderbilt.

I would also like to thank everyone at the National Security Agency who has played a part in my life. I first want to thank Brad Martin, Dr. Adam Tagert, Shelly Freitas, Healther Lucas, Stephani Askins-Yanacci, Dr. Ahmad Ridley, Dr. Bill Christian, Ray Brown, and Judy Hunt for their support and mentorship during last 5 years as a part of the Science of Security program. My research truly wouldn't of been possible without you. Furthermore, I would also like to thank Ed Divers, Paul Cote, Dr. Cheyne Homberger, Dr. Jack Stewart, Bert Head Ginny Ahalt, Jessie Byrnes, Dr. Sandya Lakkur, Donald Richardson, Katie Ahrens, Mallory McMahon, Dr. Natalie Vanetta, Dr. Christopher Smith, COL Doug McInvale Dan Rogers, Greg Smithberger, Janelle Romano, Sandy Roddy, Brianna Higgins, and Alyssa Harris for making the last 4 summer internship experiences a truly remarkable experience. I would finally like to thank members at JHU APL including Daniel Cohen, Rachel Cohen, Stephen DiBenedetto, Michael DiRossi, Malcolm Taylor, and Jacob Couch for their support throughout my summer internship experiences.

I would further like to thank everyone who was involved during my internship experiences during my

Undergraduate studies. I would most notably like to thank Dr. Robert Balog, Cooper Barry, Sarah McClung, Dr. Karen Butler-Purry at Texas A&M University, Adam Burstein and Michael Hornsby at the Executive Office of the President, White House, and Dr. Michelle Richards-Babb, Dr. Jeremy Dawson, Dr. Bashar Hamza, Dr. Anand Kadiyala, Chloe Snyder, and Kathryn Smith at West Virginia University. Their invaluable mentorship has been second to none in helping to prepare me for Graduate School.

I would like to thank everyone at UMBC for an incredible experience during my Undergraduate studies. I would first like to thank everyone at the Meyerhoff Scholarship program including Dr. Freeman Hrabowski, Earnestine Baker, Lamont Tolliver, Keith Harmon, and Mitsue Wiggs for believing in me and pushing me to be the best that I can be. I want to give special thanks to my Undergraduate research advisor Dr. Mohamed Younis for his mentorship and giving me the opportunity to learn and grow as a researcher. I also want to especially thank Dr. Yatish Joshi, and Dr. Lloyd Emokpae for their mentorship within the lab. Furthermore, I want to thank Dr. Chuck LaBerge, and Dr. Joel Morris for serving as my advisors and mentors throughout my Undergraduate studies. Finally, I want to thank all of the friends that I made along the way including Emily Schultheis DiBenedetto,Meghan Willmore, Stephen DiBenedetto, Muhed Rana, Daniel Cohen, William Heckert, Alec Pulianas, and Bethany Rohrs.

Last but not least, I want to express my extreme gratitude to my family. I first want to start with my parents Karen and Dave Potteiger, They have always believed in me, and have been with me every step of the way throughout my life. There aren't words to express how grateful I am to you guys. I additionally want to thank my brothers Tim and Nick Potteiger for always supporting me as well. I want to thank my grandparents Judy and Charles Cook, and Bill and Eileen Potteiger, as well as my relatives, Jeff Cook, Susan Boltinghouse Cook, Daniel Cook, Mercedes Cook, Evan Cook, Melissa Nilles, Dave Nilles, Andrew Nilles, and Jeremy Nilles. Finally, I want to give special thanks to close family friends including Debbie and Rob Tilghman, Mike and Lisa Flemming, Tom LeDoux, and Tim Shellum for always being there for me throughout my life.

There have been countless other people who have played a significant role in my life, and I want to express my sincere gratitude to every single one of them. I want to end this acknowledgements by thanking the National Science Foundation (David Corman), NSA Science of Security Program (Brad Martin), Air Force Research Lab and National Institute of Standards and Technology (Chee Tang, Keith Stouffer) for their support throughout my PhD studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

I.1 Motivation

In recent years there have been a number of cyber-attacks against critical cyber physical systems (CPS) such as automobiles [4]. With the change from traditionally isolated systems to the addition of remote interfaces and interconnected digital components, new avenues are emerging that significantly increase the attack surface and vulnerability sphere. The unique aspect of critical CPS with regards to the tightly coupled nature of embedded cyber devices with the physical world means that attackers can now accomplish physical damage through executing common attacks. This poses a huge threat from adversaries such as enemy states and terrorist organizations who have in the past relied on physical warfare techniques. As such, it is significantly important to secure these devices with the goal of limiting the attack surface, ensuring resiliency and reliability, and minimizing physical damage resulting from compromise.

Zero day attacks are a huge threat to CPS, bypassing defense mechanisms by exploiting a unknown vulnerabilities. As such, the strategy of locking down systems based on known vulnerabilities is no longer feasible, and a defense in depth technique needs to be undertaken to provide maximal protection. Instead of focusing most efforts on the external interface, the defense in depth strategy focuses on providing multiple layers of security, aiming to implement several backup defense mechanisms in the event that an attacker can successfully exploit a vulnerability. By implementing these techniques, manufacturers can design systems with security in mind, ensuring that there are multiple layers of security mechanisms in place.

Moving target defense techniques have been popular in the information technology domain, most commonly observed in applications such as servers. To successfully execute a cyber attack, the adversary first needs to gain sufficient knowledge of the system through reconnaissance efforts. This step often comes before the execution stage of an attack. Moving target defense techniques leverage this necessary step by periodically changing various system properties at run time, making the knowledge gained by the adversary no longer accurate. These techniques enhance the defense in depth approach by dynamically changing various system parameters such as instruction sets, memory storage locations, data representation, and network communication paths. Even though these techniques are very good at stopping cyber attack attempts, they are prone to resulting in denial of service behavior. For example, in the case of instruction set randomization when an attacker attempts to execute a payload with a wrong instruction representation, the payload will not run correctly, but this attempt will still result in the system crashing with an invalid instruction execution error. Even though this is not the desired behavior intended by the adversary, this can still be considered damaging to the CPS, especially critical infrastructure such as military applications, autonomous vehicles, and medical devices that have to have guaranteed availability. Therefore, in the CPS domain it is not just enough to prevent cyber attacks, but it is just as important to have recovery mechanisms in place to ensure that system operation will continue to be reliable with real time parameters in tact.

I.2 Research Challenges

Cyber-physical systems (CPS) are engineered systems created as networks of interacting physical and computational processes. Most modern products in major industrial sectors, such as automotive, avionics, medical devices, and power systems already are or rapidly becoming CPS driven by new requirements and competitive pressures. However, in recent years, a number of successful attacks against CPS targets, some of which have even caused severe physical damage, have demonstrated that security and resilience of CPS is a very critical problem, and that new methods and technologies are required to build dependable systems.

Semiautonomous and autonomous vehicles are a very significant domain and opportunity for CPS [5]. Modern vehicles employ sensors such as laser range finders and cameras, GPS and inertial measurement units, on-board computing, and network connections all of which contribute to a large attack surface. Recent proof-of-concept attacks on connected cars have demonstrated significant vulnerabilities and how unprepared the industry appears to be for responding to cyber-security threats [6; 7]. All potential points of compromise will need to be secured and all vehicle-related data will need to be protected.

Although a large investment and comprehensive efforts are required for addressing such CPS security challenges, there is a need for innovative methods that can provide sufficient protection given the constantly evolving nature of the threats. The high level research challenge is to develop principles and methods that ensure the development of CPS capable of functioning dependably, safely, and securely in dynamic and uncertain environments.

Design of autonomous CPS today progresses along abstraction layers usually separated into control design, software design, and platform design [8]. Control design focuses on system dynamics for satisfying stability, performance, and safety requirements, software design is responsible for functional correctness, and platform design is concerned with timing and power requirements. However, CPS security cannot be localized to isolated abstraction layers, because it emerges from cross-cutting interactions. Neglecting this fact leads to profound security failures.

Viewing challenges in CPS has both similarities and differences to the traditional information technology domain. These two domains have similar adversary models and attack surfaces. However, when looking at developing security mechanisms for each domain, different priorities need to be taken into account. For example, in the information technology domain it is acceptable to conduct frequent patching updates, and maintain basic availability of systems. However, in CPS the priorities are a little different. For example, often times CPS are required for critical tasks leading to more time and effort for taking the system offline for patching. Combined with the fact that a large majority of these systems are legacy systems, it is infeasible to conduct security from a reactive patching strategy. Furthermore, it is not just required to have basic availability, but systems need to have real time availability, where performance is not degraded and real time deadlines are consistently met. The literature has characterized the three necessary areas of research for CPS security as (1) better understanding the consequences of an attack in terms of both the cyber and physical behavior to identify adversary priorities, (2) designing novel detection algorithms to filter tampering behavior from normal physical process measurements, and (3) design resilient algorithms and architectures for taking a more proactive approach on CPS security design [9].

We take these areas of research into account when addressing the following research challenges.

- How do we protect against code injection and code reuse attacks while maintaining system availability with safe, reliable, and predictable operation?
- How do we protect legacy CPS software against non-control data attacks when source code isnt available? How do we detect variable tampering?
- How do we create predictable and reliable CPS while rapidly detecting and recovering from cyber-attacks? How do we protect against code injection, code reuse, and non-control data attacks within this architecture?

I.3 Research Contributions

To effectively protect against code injection, code reuse, and non-control data attacks, MTD needs to be utilized. The techniques such as ISR, ASR, and DSR change system parameters to decrease the effect of adversary reconnaissance efforts. It is further important to integrate all of these techniques together to broaden the scope of protection but also ensure that safe and reliable system operation is maintained. Finally, it is critical to have evaluation capabilities to measure the effect of various implemented MTD architectures on the system. It is further important to evaluate respective system metrics on hardware that is consistent to the deployment environment.

In Chapter III, we focus on the need to protect safety-critical CPS from buffer overflow based exploits such as code injection and code reuse attacks while maintaining availability. To address this problem we develop a MTD based control architecture that utilizes AES 256 ISR and fine grained ASR at the function level to protect against code injection and code reuse attacks. Furthermore, we integrate control reconfiguration in our approach to detect the presence of an attack, and rapidly transfer execution to a backup controller to avoid any disruptions in system operation. Additionally, we develop a hardware-in-the-loop testbed implementation to test the effectiveness of our approach in an environment similar to the deployment setting. Finally, we utilize an autonomous vehicle case study to illustrate the benefits of our security architecture.

In Chapter IV, we focus on the need to protect legacy CPS software from non-control data attacks in the event that source code isn't available. To address this problem we've developed a DSR approach that can operate at the binary level. First, we've developed a static analysis approach to lift the binary from an executable to an intermediate format, establish variable relationships through points-to-analysis, and create unique randomization keys for every program variable location. Second, we've implemented an attack detection mechanism that utilizes variable integrity checks based on a redundancy approach that compares two uniquely randomized copies of a variable before use to detect any instances of tampering. Since each variable copy is randomized with a different randomization key, the probability of an attacker successfully bypassing this integrity check is severely limited. Third, we've built in a reconfiguration scheme to ensure that in the event of a non-control data attack, program execution is transferred to a backup safety controller to ensure safety-critical functionality is not lost. Finally, we utilize an autonomous vehicle case study to illustrate the benefits of our approach.

In Chapter V, we focus on the need to combine the predictability and reliablity of time triggered architectures, which are commonly utilized in safety-critical CPS, with the rapid reaction and flexibility provided by event triggered architectures. Furthermore, we need to ensure that systems are protected against code injection, code reuse, and non-control data attacks while detecting and recovering fast enough to maintain safe and reliable safety-critical operation. To address this problem we've designed a mixed time and event triggered security architecture that integrates MTD techniques such as ISR, ASR, and DSR for cyber-attack protection, while leveraging benefits from the ARINC 653 architecture for supporting predictible time triggered scheduling. Secondly, we leverage event triggered functionality for designing a reconfiguration scheme to rapidly react to a cyber-attack and transfer execution to a safety-controller, all while minimizing missed deadlines and maintaining the integrity of the static schedule. Finally, we have successfully implemented and evaluated our approach using a developed hardware-in-the-loop testbed and autonomous vehicle case study to illustrate the benefits of our approach.

The overall contributions of this dissertation are described below:

- We design a MTD based security architecture to protect against code injection, code reuse, and non-control data attacks.
- We design a control reconfiguration scheme to maintain safe and reliable operation in the event of a

cyber attack.

- We design a data integrity checking approach leveraging DSR and variable redundancy to detect tampered variables.
- We develop a customized hardware in the loop testbed utilizing a combination of off-the-shelf embedded computing hardware and open source simulation software for the evaluation of CPS security experiments.
- We present an autonomous car case study to demonstrate the effectiveness of our security architecture in limiting the impact of cyber-attacks, as well as the CPS cyber and physical behavior.

I.4 Organization

The rest of this dissertation will be sequenced as follows. Chapter II introduces the respective related work in the CPS domain. The related work starts with describing the current attack surface related to the automotive CPS domain, in addition to popular exploitable vulnerabilities found in CPS. Next, real time systems background is described as well as several techniques including event triggered and time triggered scheduling. Moving target defense techniques are then described including ISR, ASR, DSR, and network randomization. Afterwards, control reconfiguration principles are described including anomaly detection, recovery techniques, and fault tolerant architectures in the literature. Finally, the related work chapter ends with a comparison between our proposed work with the work done in the literature. Chapter III introduces an Instruction Set Randomization security architecture with integrated control reconfiguration capabilities developed for protecting CPS against code injection attacks, while maintaining safe, reliable, and predictable system operation. Chapter IV extends this security architecture by integrating Address Space Randomization, and Data Space Randomization defense mechanisms for protecting against code reuse and data tampering attacks. Chapter V further extends our security approach to be compatible with a mixed time and event triggered architecture, which is a popular approach for safety-critical CPS. Chapter VI ends the dissertation with concluding remarks. Finally, Chapter VII lists the relevant publications contributing to this dissertation.

CHAPTER II

Related Work

II.1 Security of CPS

With the increasing use of CPS in modern society, the properties of safety and security are becoming more interwined, when in the past they have been traditionally treated as two independent issues dealt with by two separate communities [10]. The tightly coupled nature of CPS between the cyber and physical domains means that it is not only enough to lock down the internal data of the system, but it is equally as important to ensure proper physical dynamics of the system. As such, measuring security in CPS is heavily focused on the continuous physics dynamics, compared to the traditional discrete nature of information security. Therefore, evaluation is often conducted with complex, model based simulation environmentss [11].

In traditional information technology systems one of the the primary goals of an adversary is usually to obtain a piece of information. This has been displayed in three recent cyber attacks including a 2016 hack on the democratic national committee [12], a 2016 electronic records compromise of a large hospital chain [13], and a 2009 compromise of sensitive employee information within the Office of Personnel Management and Budget (OPM) [14]. CPS systems on the other hand have unique aspects that make both attack vectors and impact different from information technology systems. CPS have the additional requirement of fully complying with a respective specification of properties. Any failure to meet this requirements will result in a failure of the system. As such, an adversary can successfully compromise a system by disturbing the timing of real time operations, or injecting false data to lead to wrong actuation decisions. The most popular example of a CPS cyber attack is STUXNET in which hackers were able to create a worm that gained access to and altered programmable logic controllers (PLCs) to overheat centrifuges in Iranian nuclear enrichment facilities [15]. At this point, it became clear that to successfully protect CPS, a proper design needs to factor in system integrity from the security community, as well as system availability, and reliability from the safety community.

Compared to information technology systems, CPS systems often have a higher level of physical accessibility. Attackers can utilize physical attack vectors such as planting explosive devices and sabotaging railway tracks or positive train control systems [16; 17; 18] to disrupt the operability of CPS components. As such, it is just as critical to implement physical defense mechanisms in CPS, compared to cyber mitigations.

Another critical difference between CPS systems and information technology systems is that patching and frequent updates are not well suited for CPS systems due to the requirement of constant and safe operation [9].

As such, devices are often left in the field for years without updating software. Due to this fact, CPS systems often have outdated code and security measures making them vulnerable to attack [19; 20; 21]. As such, CPS security has to be designed to withstand long periods of time. Furthermore, CPS designs have to be built in a resilient fashion, under the presumption that whenever an exploit is developed defeating previously established defense mechanisms, the system will still be able to operate sufficiently, and safely.

Cyber Physical Systems have unique features and requirements that present challenges to security professionals to adapt from traditional information technology practices. The ability of CPS to interact with the environment through actuation allows for attack impact to translate from data loss to physical damage potentially resulting in fatalities. Additionally, the ability to receive information through sensors allows attackers to adjust control operation through the manipulation of sensor data. CPS provides the potential for attackers to maximize damage to their target.

II.1.1 Automotive CPS

In contrast to the analog structure of early car models, modern vehicles consist of complex systems of systems. As such, cars today are made up of hundreds of electronic digital components communicating through a mesh of interconnected networks with varying degrees of speeds, and protocols. This is backed up by [22] which states that a modern car runs approximately 100 million lines of code on 50 to 70 electronic control units (ECUs). This makes a modern car a ”computer on wheels,” presenting a vulnerability to traditional hacking methods once thought of as only applicable to computers and information technology systems.

Automotive CPS threat actors include script kiddies, political activists, insiders, white hat hackers, black hat hackers, and cyber terrorists [23]. The biggest threat comes from both cyber terrorists, and black hat hackers as these adversaries often have the most malicious intentions in causing bodily harm, and damage. Additionally, these groups of adversaries are often state sponsored so they are very sophisticated and resourceful in their strategies. Researchers have to protect against this adversary group first as it is no longer hard to picture an attempted assassination attempt or terrorist attack through hijacking a car. However, with increased academic research comes more open source tools, and documentation. Therefore, lower skill adversaries such as hobbyists, and coders are gaining the potential to experiment and develop software for vehicle infrastructure. As such, these groups also have to be taken into account in the future when developing cybersecurity design for vehicle infrastructure.

Common vehicle systems include the electronic control units (ECUs), telematics control units, keyless entry system, TPMS system, radio system, and airbag control units [7]. The architecture of a car is grouped into three categories, all of which contain their own vulnerabilities. These categories include:

- **Electronic Control Units** - The embedded computer devices that control the various internal functions

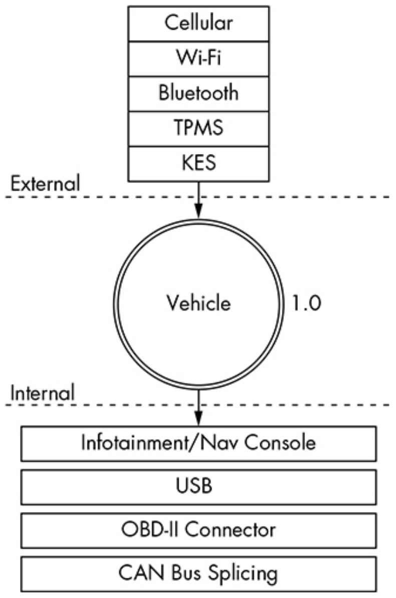


Figure II.1: Vehicle Attack Surface [1]

of a car.

- **External Interfaces** - The method of external entities interacting with the car.

- **Internal Network** - The network that the internal electronic control units communicate on

A high level threat model of these interfaces is illustrated in Figure II.1.

An ECU is an ”embedded system that controls one or more electrical systems or subsystems in a vehicle.” These devices serve as the brains behind the modern car, taking in data from onboard sensors, performing calculations, and distributing instructions to the various in vehicle electronic systems to maintain proper and efficient driving and operational performance. ECUs govern practically every aspect of vehicle functions from small tasks such as activating brake lights or opening windows to critical functions such as autonomous brake systems. Each ECU typically works independently operating its own firmware, but complex tasks may require cooperation among multiple ECUs [24].

External interfaces are means from which outside actors can communicate to the vehicle. From Figure II.1, the most common external interfaces to the car include cellular, wireless (wifi), bluetooth, tire pressure monitoring system (TPMS), and KES (Key Fob). Cellular communication channels are utilized for telematics system operations. Telematics services are widely utilized for remote vehicle monitoring especially in the case of routing emergency response, or predicting future maintenance needs. The most common services are OnStar [25] and Lojack [26]. OnStar is a remote monitoring company that provides customers a line of assistance in the case of emergency. To accomplish this task, OnStar requries several sensor values such as speed, collision detection, and GPS coordinates among others. Therefore, in the case of an accident, the OnStar monitoring center can automatically detect an emergency and alert the authorities of your exact location to respond to. Lojack is a device that tracks the GPS coordinates of your car in the case that it becomes stolen. Furthermore, the device utilizes a cellular network to communicate its precise location in real time to the police. Many newer model cars have wireless interfaces that allow mobile phones, tablets, and computers to control specific features. These features range from controlling windows, and door locking, to controlling the infotainment center, to seat control. This wireless feature further serves as a wifi hotspot, for each passenger to have the capability to utilize the internet from inside the car. Bluetooth is most commonly utilized in cars as a short range communication protocol for mobile phones to connect to the infotainment center. This connection is used for transferring audio such as music or phone calls between the vehicle interface and phone. The tire pressure monitoring system uses 4 sensors on each respective tire to record the real time tire pressure while indicating a warning signal in the case that one of the readings is below a threshold. Each external tire pressrue sensor communicates to a central vehicle ECU through a short range Bluetooth communication protocol. Finally, a vehicle key fob is actually an active RFID device that is validated from the car through a short range RF communication channel.

The final category is the internal interface. These are devices that connect directly into the internal vehicle network. Examples include the infotainment console, USB devices, and the OBD-II connectors. The attack surface of these devices requires physical access to the inside of the car. However, this category of devices serves as the most critical threat to the security of the vehicle, as once these devices are compromised, there are very limited precautions that prevent them from controlling critical ECUs of the car. Newer model vehicles consist of USB ports designed for charging mobile phones, and media players, while serving as a direct interface to the infotainment media player. Due to this direct connection, malware injected on the USB drive can infect the infotainment center and lead to a compromised state of the network. The next critical internal interface is the infotainment center. The infotainment center serves as the central entertainment hub of the car. These devices include a radio interface, phone call interface, music player interface, GPS interface, and other applications. These applications communicate with ECUs through the internal network as well as external entities through short and long communication protocols. Therefore, an obvious path to compromising these devices is through the external communication interfaces. However, it is important to note that third party applications can also be installed on these devices. As such, a new emerging threat is infecting the infotainment center through the use of third party applications. The infotainment center is the most vulnerable remote attack avenue for the vehicle due to the vast connections it has with the external environment. The final internal interface to mention is the OBD-II port. The OBD-II port is a circular input port located under the steering wheel key slot that serves as a direct connection for maintenance personnel to interact with the internal vehicle network. Usually, when a car is serviced from a dealership, personnel connect to the OBD-II port to gather diagnostic information such as emissions, engine temperature, average speed, and other details. This direct connection allows for the injection of packets directly onto the internal network, versus through intermediary paths such as with the infotainment and USB interfaces. Therefore, by compromising a maintenance person, or obtaining physical access to the car, adversaries can directly inject malware onto the internal network and consequently infect the connected ECUs.

The internal automotive networks consist of a series of multiple communication buses with varying protocols. The most common communication buses utilized are the Controller Area Network (CAN) Bus, the Local Interconnect Network (LIN) Bus, the FlexyRay Bus, and the Media Oriented System Transport (MOST) Bus [27]. These buses are described below:

- **CAN Bus** - Most common communication bus. A serial bus designed from two twisted wires transmitting a high and low voltage respectfully. Data rate peaks at 1 Mb/s. CAN is responsible for many systems such as vehicle control, safety, and electrical systems. It operates like a broadcast network in which packets are conveyed to all nodes on the network and it is the responsibility of the nodes to determine whether or not to process them. Each CAN packet has an identification (ID) field to help determine which nodes will process it.

- **LIN Bus** - A single wire subnetwork for lower bandwidth, low cost, and low end multiplexed communication in automotive networks. LIN uses a master-slave model, where a master node and up to 16 slave nodes share a bus. The master sends out messages to all slaves, and the slave can only send a message if requested by the master. The communication rate can peak at 20 kb/s. Since this protocol is used for low cost, low criticality elements, it is used for controlling comfort elements such as electronic window lifts or windshield wipers.

- **FLEX Ray Bus** - The successor to the CAN protocol offering speeds up to 10 Mb/s. The high speed and reliability enable the use of X-by-Wire technologies such as the electronic control of currently mechanical control systems such as steering and braking. This bus is widely used for vehicle safety systems, but due to its high cost, is limited in its widespread use in the internal network.

- **MOST Bus** - A synchronous network used to transmit multimedia data through the car via optical fiber. Multiple data channels are offered as well as a control channel. Synchronous communications are used to transfer streaming data such as audio or video signals, while asynchronous communications are used for scenarios such as retrieving data from the internet. Speeds can reach up to 24 Mb/s. The MOST bus is used for the infotainment center and entertainment applications in the vehicle.

In a 2014 Audi A8 vehicle most of the networks consist of varying CAN buses including the drivetrain, distance control, gateway, and convenience systems [28]. Additionally, some convenience systems such as the keyless entry, power windows, and windshield wipers use the LIN Bus due to the lower bandwidth requirements. Finally, the MOST bus connects the infotainment system of the car to the rest of the network.

There are several vulnerabilities with the CAN Bus packet structure and protocol dealing with confidentiality, integrity, availability, authenticity, and non-repudiation [27]. From a confidentiality standpoint, every message sent on the CAN Bus is broadcast to every other node connected to the bus. Due to this behavior, if an attacker were to compromise one connected ECU device, they could ease drop on the rest of the communications on the CAN bus, and also spoof messages to any connected ECU. Therefore, this type of communication is truly only as strong as the weakest link as once one node is compromised, the whole network is subject to being compromised. In regards to vulnerabilities with integrity, the cyclic redundancy check (CRC) in the CAN packet to check if a message has been modified. However, it is easy to forge a correct CRC so this is not sufficient for preventing an attacker from creating false messages or modifying existing messages. In regards to availability, since every packet has a user defined priority bit, an attacker can spoof numerous high priority packets to flood the CAN bus and prevent proper communication between the ECUs. In regards to authenticity, the packet structure of CAN does not include a field for authenticating the sender of the message. This makes it very easy to spoof messages on the network. Finally, from a non-repudiation standpoint there is no method in the protocol for a node to prove that it has not sent or received a given message.

By exploiting these vulnerabilities, several types of attacks can be executed on the network. [29] describes 5 types of potential attacks including data stealing, control overriding, vehicle degradation, data falsification, and external sensor attacks. Data stealing deals with an attacker connecting to the CAN network, and eavesdropping on the rest of the ECU communications. Control overriding deals with an attacker taking advantage of the susceptibility of CAN networks to denial of service attacks by injecting high priority messages on the bus which would always be executed over the normal control messages. As such, an attacker could use this to take control and hijack vehicle operations. Vehicle degradation deals with an attacker spoofing messages about the current condition of the vehicle to trick the system into running inefficiently or become damaged by overheating, running out of gas, or having a flat tire. Data falsification deals with relaying false information to the driver to result in unsafe behaviors such as disabling an airbag warning light when the airbag system is dysfunctional.

Putting these concepts into practice, researchers in [30] have shown the ability to effect the behavior of vehicles through attacks on the CAN bus. The research was built on the assumption that adversaries could gain access to the CAN bus through external means, but once access is gained atacks could be executed to turn the horn on, shut the engine off, shutting off the brakes, disabling steering, spoofing the speed on the speedometer, and increasing the amount of distance shown on the odometer. Additionally, the same researchers were able to reverse engineer and exploit a modern Jeep Cherokee in [31]. Even though this research was conducted in the purely academic context, it is not hard to imagine the applications that a malicious adversary could use these concepts for. As such, it is critically important to increase the amount of security mechanisms in these systems, and bring cybersecurity strategies into he earliest stages of vehicle design.

Previous vehicle security efforts have focused on hardening internal ECU communication, implementing anomaly and intrusion detection systems, and securing external communication interfaces. For securing the internal vehicle communication, a majority of research is focused on the CAN bus protocol due to being the most widely utilized protocol in vehicles [32]. The CAN protocol at the basic level is vulnerable to masquerade, and replay attacks due to the lack of authentication. There has been progress in authentication, such as a Message Authentication Code (MAC) technique in Tesla vehicle models [33], as well as a time triggered implementation proposed by [34] for establishing global time. However, the biggest challenge is developing authentication mechanisms with low enough overhead to be feasible in deployment environments due to low bandwidth of the CAN protocol. The second area of security research focuses on securing the underlying processes and computations operating within the vehicle. Approaches focused on being preventative and reactive in accordance with protecting systems from attack. [35] proposed to introduce symmetric encryption techniques to secure the integrity of ECU computational processes, while hashing techniques can be utilized to secure data. Additionally, secure software engineering practices have gained more attention, to minimize the amount of attack vectors available in ECU's based on bad coding practices [36]. Furthermore, to address attack detection and reaction, research has applied traditional information security techniques such as honeypots, and intrusion detection systems to the vehicle domain [37]. By leveraging artificial intelligence, and machine learning, these techniques can be optimized to minimize the amount of time, and accuracy of detecting an adversary event. The final area of research focuses on securing the internal vehicle network from external devices [38]. By monitoring third party apps, connected phones and media players, approaches have focused on implementing firewall techniques within the vehicle gateway interface. As such, by designing only one path of entry through this gateway components, these techniques have been proven effective to block malicious entities from gaining access to the internal, more safety-critical vehicle components. When analyzing the related research in vehicle security, a lot of the approaches focus on locking down the system from external threats. For example, authentication ensures that attackers can't spoof internal communications, while firewalls serve as an outer layer of protection, blocking malicious entities from gaining access into the internal network. Additionally, to appropriately optimize these types of defense mechanisms, the designer must have full knowledge of the potential vulnerabilities, as well as every location an adversary can potentially infiltrate. However, it has been commonly shown that researchers and adversaries can find ways around defense mechanisms in place, whether due to lack of sufficient protections due to designer ignorance, or by introducing zero day exploits not previously known in public [31]. As such, it is not feasible to only rely on locking down systems based on previously known vulnerabilities. A defense in depth approach is the best option for optimizing the security of a system, due to implementing multiple layers of protections. MTD along with control reconfiguration enables a backup defense protection mechanism that can stop attacks such as code injection, code reuse, and data tampering, in the event that an adversary has defeated outer defense protections. Additionally, be introducing control reconfiguration capabilities, resilience can be designed into systems to maintain availability during attack sequences, when in regular defense mechanism implementations, an attack would either result in system hijacking or crashing.

II.1.2 Buffer Overflow Vulnerabilities

One of the most popular vulnerabilities are buffer overflows, which have been regarded as the most commonly used exploit over the last several years [39]. Buffer overflow attacks are widely publicized in the traditional information technology domain for the purpose of network penetration, most publicized as an avenue for executing sql injection attacks on web servers. However, these vulnerabilities become more interesting in the Cyber-Physical domain, where they are commonly utilized to exploit memory allocation functions in low level programming languages such as C and C++. In addition to providing the capability for exfiltrating stored data, attackers can also alter the control flow through non bounded input to execute malicious code on the embedded controllers. With previous reconnaissance on the control API, attackers can reverse engineer controller instructions to spoof new instructions to alter the physical behavior of the embedded device. One possible result of a buffer overflow vulnerability is a code injection attack. As such, the attacker can divert the program control flow to inputted malicious code, which will consequently be allowed to run unchecked on the system. The buffer overflow and code injection attacks are described below.

Buffer overflows have been the most common form of security vulnerability over the last ten years, and are additionally widely used in the area of remote network penetration. This vulnerability is often times the first step for an attacker to gain entry into a host system without having physical access to the geographic location. As such, the attacker usually attempts to target a root program with this type of vulnerability to subvert the function to their respective code. With successful execution of this attack, instructions can be run with root privileges on the machine.

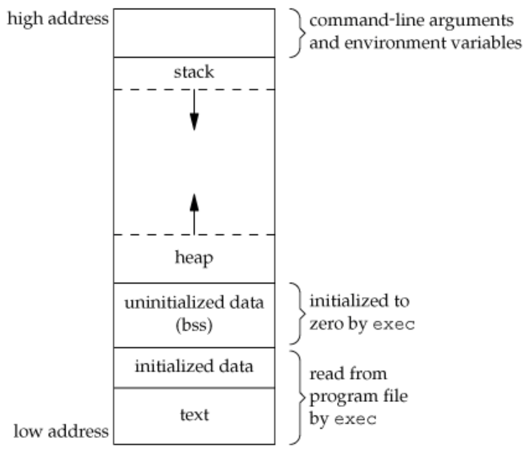
To understand how a buffer overflow attack works, we first have to understand the process memory storage of a program, and how inputted data is stored in that structure. This tutorial is based off of the tutorial in [40].

```c
#include <stdio.h>
void secret()
{
    printf("This is the secret function\n");
}
void normal()
{
    char buffer[10];
    printf("Input:\n");
    scanf("%s",buffer);
    printf("This is the normal function\n");
}
int main()
{
    normal();
    return 0;
}
```

In this program there are three functions:

- **Main Function**: Is the starting point and main process of the program
- **Normal Function**: Stores an inputted string into a 20 character buffer array. Is called from the main function.
- **Secret Function**: Prints a congratulations message. Is not supposed to be called from the program

Under normal operating circumstances, the expected output of the program is a message stating this is the normal function. However, because the inputted string function scanf does not have a bounding capability, if a user enters more than the 10 allocated characters for storage in the buffer variable, some interesting behavior occurs.



high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Figure II.2: Process Memory Layout

In a running process, the allocation of memory appears consistent with Figure II.2. In this process, the memory is allocated with 6 different partitions.

- **Command line arguments** - Parameters that are passed to the program before execution begins
- **Stack**: Contiguous block of memory where all of the function parameters, and local variables are stored. It is a last in first out (LIFO) data structure meaning that the last variable added to the memory block is the first to be removed. The memory grows downward so the first item added will be at the highest address of the stack. The last item added will be at the lowest address of the stack.
- **Heap**: Contiguous block of memory where all of the dynamically allocated memory is stored. The heap grows upwards towards the stack so the first item allocated will be at the lowest address and the last item allocated will reside at the highest address.
- **Bss Segment**: Where all of the uninitialized data is stored. This segment consists of all of the global and static variables which are not initialized in the beginning of the program.
- **Data segment**: This segment contains all of the initialized variables in the program
- **Text segment**: This segment contains all of the executable code for a program

Now, Figure II.3 shows how the stack will grow in the memory when the echo function is called from the main function.

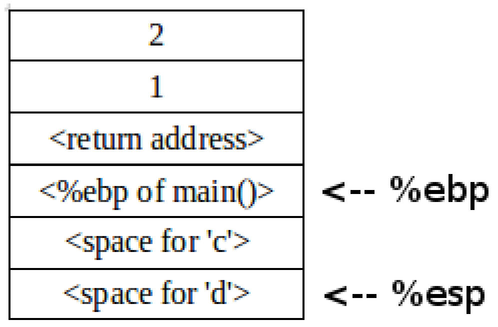


Figure II.3: Stack Memory Allocation

EBP represents the base pointer or the end of the respective stack memory allocation for the main function. ESP represents the stack pointer which is the allocated stack memory for the current function (echo). As such the initialized variables in the echo function will be allocated downwards from the base pointer to the stack pointer. The allocated variables above the base pointer are stored from the main function before the echo function is called. It is important to note that the return address is allocated on the stack before the echo function is called. This is the vulnerability of the buffer overflow attack. When the user string is inputted into the 20 character initialized buffer, if the input is longer than 20 characters (bytes), the input will proceed to overwrite the adjacent upwards memory. If the user string is long enough, it will overwrite the return address of the function, telling the echo function where to return to after completion. As such, if the user string is crafted in a way that a hexadecimal value address occurs at the exact spot to overwrite the function return address, the program control flow can consequently be controlled by the user and the echo function will return to the user hexadecimal inputted address.

The buffer overflow attack consists of two steps. These steps are illustrated in Figure II.4.

1. Arrange for suitable code to be available in the program's address space and is executable

2. Get the program to jump to that code, with suitable parameters loaded into the registers and memory

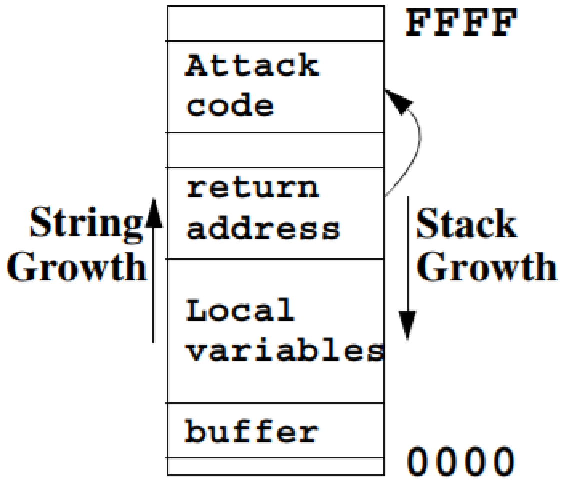


Figure II.4: Buffer Overflow process flow

Therefore, once the user can overwrite the function return address with the specially crafted input string, they can either:

- Return to already existing code in the program
- Return to the beginning of the user string which will be crafted with user specified binary instruction code

For the first option, the user can set the return address to be another location in the program code. This is referred to as a code reuse attack. In code reuse attacks, attackers edit the program control flow to return to a sequence of already existing code in the program. Code reuse attacks are useful in cases where the stack is non-executable, preventing an attacker from running instructions from input on the stack. Code reuse attacks have been successful for privilege escalation in the Android operating system, creating rootkits, and injecting code into Harvard architectures [41]. In the case of our example, the function secret function is never called by the normal program execution. However, if the attacker were to change the return address of the echo function to the beginning of the secret function, this function will then execute and the output will contain the statement ”Congratulations you have entered the secret function.” In this example, the consequences are trivial as a print statement will not really have a devastating effect on the program, and a software developer will not make extra functions with malicious code. However, an attacker can use sequences of instructions to accomplish their goal. For example, one of the most popular attacker goals is to be able to open a command shell [42]. In this case, the return address of the echo function on the stack can be changed to the starting address of the System() function in the system library. The inputted string can additionally be crafted in a way to allocate the string '/bin/sh' onto the stack as a parameter for the system function. As such, when the program is executed, the echo function will return to the system function with the command shell executable as a parameter. This allows the attacker to remotely open a shell and execute arbitrary commands on the system.

For the second option, a code injection attack can be used. In this case, an attacker can push arbitrary code onto the stack through a buffer overflow. Then the attacker can subsequently overwrite the stack return address to redirect control back to the injected attacker code [41]. Code injection attacks are used in cases where the stack has executable permissions. For code injection attacks, the process involves inputting the code to run in the input string in a hexadecimal format. This part of the input is called the payload, which is the code that will be executed. The end of the input string will be crafted to overwrite the function return address to be the beginning of the payload section of the input. As such, after the input is inserted into the program, control flow will return from the current function to the beginning of the attackers payload code. After this point, the payload code will be executed. Normally, due to restrictions on the size of an input string and consequently the size of the payload, the injected code needs to be small in size. It is common for attackers to inject the system function binary code as a payload to be executed, consequently opening a command shell for further commands to be executed on the system. The goal of using the return-to and code injection attacks is not to insert all of the desired instructions into the exploit, but to use these avenues for establishing a foothold into the system for further command execution.

II.2 Moving Target Defense

With attackers becoming increasingly sophisticated, cybersecurity is becoming less focused on completely locking down systems and more focused on decreasing the risk of attack to the system. If attackers have the motivation, time, knowledge, and resources to attack a system, eventually they will be successful in gaining entry. As such, attack risk is lowered by decreasing attacker motivation and the knowledge of the system. Moving target defenses are a good tool for decreasing attacker knowledge by constantly changing various system properties while preserving essential semantics, executing a security through diversity strategy [43]. Therefore, even if an attacker were to obtain knowledge of system vulnerabilities at one point in time, these vulnerabilities wouldn't necessarily be true during future time periods. Moving target defenses have long been applied to the traditional information technology domain, but however are fairly new to the cyber physical system domain.

Moving target defenses can be characterized into 5 sections: dynamic runtime environment, dynamic software, dynamic data, dynamic platform, and dynamic networks [44]. These categories can be characterized in regards to their place in the execution stack and are described below [2]. Figure II.5 presents this

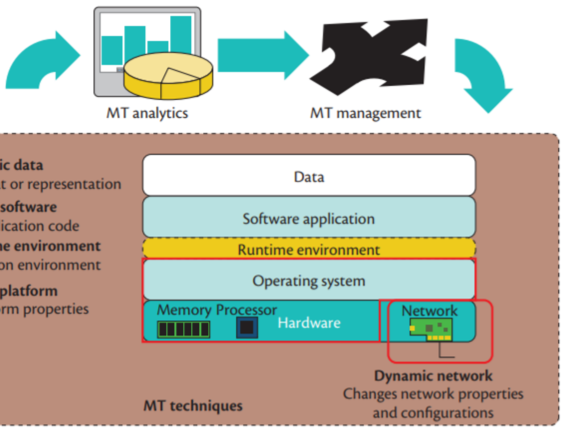


Figure II.5: Moving Target Defense Execution Stack [2]

organization.

Dynamic runtime environment techniques focus on changing the environment present by the operating system during execution. These techniques aim on preventing attackers from exploiting software vulnerabilities to compromise a system. Two techniques associated with this category consist of instruction set randomization (ISR), and address space randomization (ASR). Instruction set randomization involves changing the format of the actual program instruction opcodes dynamically during execution. This prevents the attacker from predicting and injecting code payloads into the system. The second technique, address space randomization, involves changing the virtual memory layout of the program. This prevents the attacker from rerouting the program to run code at certain locations in memory, and to assume adjacent variables to overwrite in the event of a buffer overflow attack.

Dynamic software involves changing the application code such that the attacker can no longer guess the internal behavior of the program based on fuzzing inputs. The most common technique for this category is diversification. Diversification involves rewriting program code in different formats such as with different instruction sequences, reordering functions and instructions, and reordering the internal data structures in such a way to accomplish the same functionality with a different process. The most common use of these techniques is to prevent side channel and fuzzing attacks on the program.

Dynamic data involves changing the representation of application data such that unauthorized use is hindered but the program semantic use remains the same. The most common technique in this category is data encryption. Data encryption seeks to guarantee that even if attackers were to access data from a program, they would not have the means to figure out the true context of that data.

Dynamic platform involves changing the properties of the computing platform in an effort to disrupt attacks that rely on specific platform characteristics. Some platform characteristics include operating system,

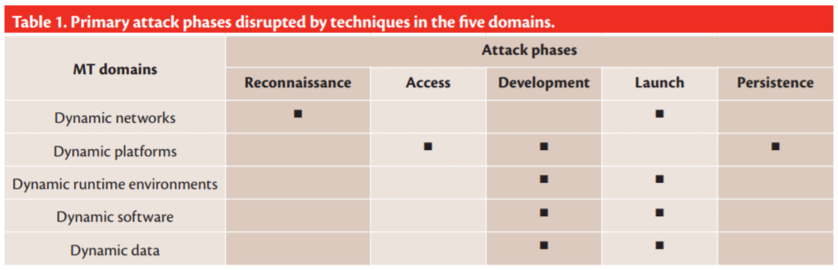


Table 1. Primary attack phases disrupted by techniques in the five domains.

MT domains	Attack phases				
	Reconnaissance	Access	Development	Launch	Persistence
Dynamic networks	■			■	
Dynamic platforms		■	■		■
Dynamic runtime environments			■	■	
Dynamic software			■	■	
Dynamic data			■	■	

Figure II.6: Moving Target Defense Attack Phase Prevention [2]

processor, and communication means. It is most common for this technique to involve migrating applications between different machines to prevent attackers from targeting a specific platform or vulnerability of the system.

Dynamic networks focuses on continuously modifying network properties to lower the probability of success for network born attacks. This category often involves changing ip addresses and network ports through software defined networking means, but also includes changing communication protocols, and changing the topology of the network.

Each of the moving target defense techniques described above is designed to disrupt a specific phase of the attack sequence. To understand this further, the stages of the attack sequence are described below [2]:

- **Reconnaissance** - Attackers find and collect basic information on their target. Examples include spear phishing or determing a target hosts IP address using network scanning.
- **Access** - Attackers take actions to collect detailed information on their targe. Examples include probing against a server to determine its architecture, operating system, and configuration. This stage produces the vulnerabilities that attackers can leverage to access the target.
- **Development** - Attackers research and develop an attack that can exploit a vulnerability on the target.
- **Launch** - Attackers deliver the attack payload to compromise the target. The payload can be delivered through a variety of means including the network, infected media, or malicious executable code.
- **Persistence** - Attackers insert a backdoor on the target to ensure future access.

The respective association between moving target defense category and targeted attack phase is observed in Figure II.6.

For this proposal, the dynamic runtime environment category will be looked at extensively focusing on applications of instruction set randomization, address space randomization, and data space randomization in cyber physical systems. Several attacks that moving target defenses address include data leakage attacks, resource attacks, injection attacks such as code injection and control injection, spoofing attacks, exploitation of authentication, exploitation of privilege and trust, and supply chain or physical security attacks [44]. In regards to instruction set randomization, address space randomization, and data space randomization, this proposal looks specifically at preventing code injection, and control injection attacks.

II.2.1 Instruction Set Randomization

Many techniques have been proposed to defuse code injection attacks including stackguards, memory management unit access control lists, control flow integrity, and masking code pointers [45]. Additionally, a sytactic checker has been proposed for Sql based applications [46]. However, instruction set randomization is the most widely accepted technique for preventing these types of attacks. Normally, code injection attacks utilize buffer overflow vulnerabilities for input processing in low level languages such as C, and C++ to insert malicious code into the program and divert control to execute that code. However, it has been demonstrated that other scripting languages such as Web CGI, bash, and SQL have been vulnerable to attacks [47]. For code injection attacks to be successful, the adversary has to rely on knowing the native architecture of the running program code on the target machine. However, ISR prevents this knowledge by changing the code architecture to a custom, randomized version that is not publicly known. For the code injection attack to now be successful, the attacker now needs to know the randomization key.

CPU instructions for common architectures such as arm, and x86 have two parts, the opcode and the operand. The opcode defines the operation to be performed while the operands define the arguments to the function. It is important to remember that these instructions are at the binary level so the combination of opcode and operand will become a 32 bit binary sequence on a 32 bit platform and 64 bit binary sequence on a 64 bit platform. During ISR implementations, we can create new instructions binary sequences by using cryptographic algorithms to create new mappings between functions and opcode and operand combinations. The most common algorithm used in the literature is using a XOR command to change the binary sequence with a randomization key. For example, for the attacker to now guess the randomization key, it will take 2^{32} attempts on a 32 bit platform and 2^{64} attempts on a 64 bit platform. Attempts have been made in the literature to guess the ISR randomization key using various techniqeus such as side channel attacks, plain text key communication, and exhaustive key guessing [48; 49]. However, other more advanced encryption algorithms such as the advanced encryption standard (AES) can be used for more security.

ISR implementations can either be hardware or software based. For hardware based implementations, a customized processor or FPGA needs to be used to insert derandomization functionality into the processor pipeline. Researchers have successfully used OpenSPARC FPGA processor to create a hardware based ISR prototype [50; 51]. For early stage implementations of software based ISR, emulators such as the Bochs x86

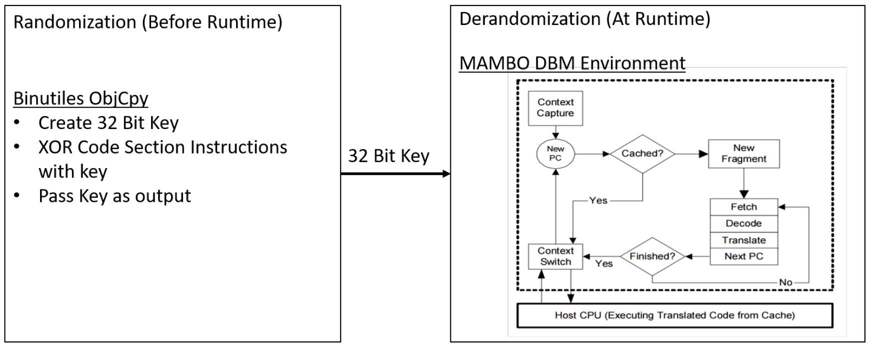


Figure II.7: Randomization Architecture

emulator were used for the purposes of processing customized instruction architectures [52; 53]. However, for more recent versions of ISR software implementations, there is a push for the use of dynamic binary translators which can be used to create a virtual layer between the execution program and processor pipeline. Several software implementations exist based off of DBTs including PIN [54], and STRATA [55] for x86 processors along with MAMBO [56] for Arm based processors. With the recent boom of arm based embedded devices in the internet of things, MAMBO will become more relevant in the future. Figure II.7 illustrates the architecture of an Instruction Set Randomization Framework utilizing the Mambo dynamic binary translation tool.

For the ISR implementation there are two stages: the randomization stage where the binary executable is statically randomized and the derandomization stage where the instructions are reformatted back to a valid architecture for execution by the processor. For the Mambo implementation the Binutils objcpy() function is customized to randomize each 32 instruction block of an ELF executable file by utilizing a XOR symmetric encryption function with a user defined randomization key. This randomization key will then be passed to the Mambo environment for the derandomization process. Mambo is developed to operate by switching contexts between the host CPU and mambo virtual layer which is operating in the same application space as the target executable program. Mambo creates objects called code blocks which are combined in a code cache which is executed in the native host CPU environment. Each code block is developed by iterating through individual code instructions of a target program until a control flow operation is reached. These operations include instructions such as jump, branch, and conditional branch instructions that transfer to other parts of a program. During this iteration step, each instruction fetched will be derandomized by conducting another XOR operation with the stored randomization key. Each code block will then be assigned attributes such as a starting address for the program counter, and the size of the code block. At this point the context will be switched to the host CPU and the code block instructions will be executed natively, at which point the new program counter address will be received after the final control flow instruction. At this point, the process repeats itself with instruction iteration, code block development, and code block execution. Once a code block is developed for code that is jumped to, the old code block will create an address pointer to the start of the new code block, resembling the new address to jump to.

II.2.2 Address Space Randomization

An overwhelming amount of security advisories from US Cert have described memory corruption attacks as the top major risk to systems, enabling attackers to execute remote code on critical systems [57]. In a majority of these cases, especially for executing code reuse attacks, the attacker needs to have knowledge of a specific address location for the target code to be run. Address space randomization (ASR) attempts to leverage this necessary piece of information by introducing artificial diversity in which various segments of a program and system. This can be implemented in multiple degrees of granularity ranging from randomizing the starting base addresses of shared libraries and program segments to fine tuned randomization of individual lines of code in a program [58]. By changing the location of various program segments, external memory access becomes unpredictable, and the attack will result in an invalid memory address being accessed. ASR can be implemented by randomizing a subset or all of a program parameters such as the base address of the stack, base address of the heap, order of static variable,s addresses of function call targets, base addresses of shared libraries, and the order of functions in a shared library [59].

There have been developed ASR implementations in the literature on Linux [60], Windows [57], Macintosh [61], and mobile platforms [61]. On the Windows ASR implementation, particularly on the Windows Vista platform, it is noted that there are a couple of limiting factors to randomization such as insufficient ranges of randomization, incomplete memory randomization, and limited documentation about existing randomization algorithms. This makes it more important to evaluate the extent of ASR implementations in the newer Windows 8, and 10 versions. The Linux versions of ASR referred to as address space layout randomization (ASLR) seem to be further along with more open documentation, as well as more user access to randomization customization parameters in the kernel. However, it is noted that on 32 bit Linux systems there is only 16 bits of randomization and on 64 bit Linux systems there are 32 bits of randomization [62]. ASR is implemented in Macintosh computers starting at the OSX release. However, these versions seem somewhat limited with only providing the ability to randomize the base addresses of shared libraries. It was noted that in the process of implementing ASR on mobile platforms there have been discovered problems such as the default shared library prelinking of the Android operating system, and the read only access of the file system. However, researchers were able to create a workaround called retouching to randomize prelinking libraries without needing to modify the kernel.

There have been several demonstrated attacks against ASR implementations. One of these attacks includes a timing attack that can exploit a hardware extension called the Intel Transactional Synchronization Extension (TSX) to break randomization in Windows, Linux, and MAC computers in under a second with near perfect accuracy [63]. Additional attacks taking advantage of return oriented programming and relative jumping addresses to target code addresses [59]. However, it has been noted that advances in position independent code compilation and self randomization algorithms have now mitigated some of these attacks.

II.2.3 Data Space Randomization

After obtaining access to a program through buffer overflow vulnerabilities, attackers can manipulate non control program variable data in an integrity attack to alter program behavior without altering control flow. One common technique utilized to alter these types of attacks is data space randomization (DSR). Data space randomization changes the internal or external representation of an applications data in such as way as to ensure that the semantic content is unmodified but unauthorized use, access, or modification is hindered [2]. For this to be accomplished by randomizing the format, syntax, encoding, and other properties of the data. As such, DSR acts similarly to ISR in using a key based randomization and de-randomization process to encrypt variable data sensitive to attack. Each variable data object is randomized before they are written to memory and is derandomized after they are read from memory. Additionally, DSR provides a much larger range of randomization than ASR, allowing for all 32 bits to be used on a 32 bit platform compared to the 16 bits used on Linux ASLR implementations. Consistent with the ISR process, the randomization process can be accomplished by using an XOR operation with a randomization key [64; 65]. In these implementations the overhead is minimal with an average performance overhead of around 15% [66]. Additionally, there is also the possibility of using other symmetric encryption algorithms such as those in the AES family to add further security to the application. DSR provides both the ability to use a common shared randomization key, but for enhanced security, each variable should be mapped to a unique randomization key. Implementations of DSR started with a software toolkit called PointGuard. [67]. Pointguard randomized the stored pointer addresses to prevent attackers from gaining reconnaissance knowledge about pointer data. However, current DSR implementations now not only randomize pointer addresses but also the stored variable data. Some attacks against DSR listed in the literature include data leakage attacks, brute force and guessing attacks, and partial pointer overwrites [64]. However, strategic derandomization, high randomization entropy these types of attacks are deterred.

II.2.4 Network Randomization

For attackers to identify targets and vulnerabilities through reconnaissance, they often rely on known information about the connected network. The static nature of current networks makes reconnaissance easy, allowing for attackers to maintain privileged access for a long time once a vulnerability is discovered. This is especially significant as the internet task force has declared a number of attacks that can be implemented with an attacker correctly guessing a combination of transmission control protocol (TCP) attributes including the protocol, source address, destination address, source port, and destination port [68]. Furthermore, this leaves networks open to attacks from worms, especially hitlist worms who have preprogrammed lists of target ip addresses and entry ports to use for infection and spreading [69] However, the concept of network randomization seeks to continuously modify various network attributes such as addresses, ports, protocols, and logical network topology to deter the attacker from gaining relevant information necessary to conduct network borne attacks [2]. Instead of focusing on hardening a system which is traditionally done in information security, this technique focuses on reducing the risk of attack by increasing the exploration space and changing the attack surface [70]. With the advancements in software defined technologies, these techniques are becoming easier to implement.

Network randomization implementations in the literature consist of a combination of randomizing network ports, ip addresses, and network paths. A software prototype implementation has been developed combining the capabilities of the above three randomization features [71]. To change the assigned ip address and port, the iptables utility of the netfilter kernel module is utilized. Additionally, software defined networking controllers can be used to change between various routes between network nodes, preventing attackers concrete knowledge of communication paths. By changing these network features, the attacker exploration space is increased, reducing the probability of a successful attack, and preventing an attacker from relying on previous gathered reconnaissance information.

II.3 Control Reconfiguration

II.3.1 Attack Detection

Due to the tightly coupled nature between the cyber and physical domains in CPS, there are unique aspects that need to be considered with regards to anomaly detection that are different than traditional information technology techniques. As attackers can compromise cyber system aspects through physical attacks, and physical aspects through cyber attacks, the interactions between these domains need to be considered. As such, anomaly detection in CPS focuses on both the cyber and physical layers of a system.

In information security, cyber-attacks are detected through suspicious activity referred to as anomalies. Anomalies are defined as patterns in data that do not conform to a well defined notion of normal behavior such that any presence can be correlated with the presence of a cyber attack or system malfunctioning [72]. As such, it is important for present systems to have accurate detection mechanisms not only to identify the presence of abnormal behavior, but also to take intervention measures to respond to the situation appropriately to bring the system operation back to normal. Anomaly detection in the cyber domain revolves around monitoring behavior relating to network traffic, and system operation. This detection process can include both human in the loop processes [73], as well as fully automated processes.

In the Cyber-Physical domain, attack detection is not only focused on detecting anomalies in cyber behavior, but is also concerned with activity that poses a threat to the overall system availability, and reliable functioning. In CPS, safety, availability, and reliability are the most important parameters of design. As such, any activity that can potentially lead to system crashing should be defined as a priority. For the purpose of detecting attacks in this manner, the underlying operating system mechanisms can be leveraged. In most operating systems there are kernel signal exceptions that are triggered in the instance of an unsafe state. For example, if a program attempts to access memory from a restricted location, an ”invalid address access” signal will be generated. Additionally, programs have access to API functions that can check to see if any of the underlying kernel signals have been generated. These calling functions can either be event based, or polling based functions. In either case, it is the responsibility of a successful recovery, to be able to utilize the API functionality to check for various generated operating signals triggered from an unsafe state in the program, and then perform appropriate measures to resolve the issue. By performing this detection process as fast as possible, the system down time can be minimized, and the system availability can be maintained in a reliable, and safe manner.

II.3.2 Recovery

CPS are considered real time systems, relying on system operation with fixed deadlines to process information and execute control algorithms in a timely, predictable, and reliable manner. There are 2 categories of real time systems: hard real time systems and soft real time systems [74]. Hard real time systems include tasks that are critical to the functionality of system operation. If one of these tasks were to fail to meet their respective deadlines, then the system would no longer be able to function. Examples of these types systems are often associated with safety critical systems such as medical devices such as pacemakers, defense applications such as missile guidance systems, and nuclear systems where any failure could lead to devastating consequences. The next category of real time systems is soft real time systems. These types of systems include tasks where a failure to meet an associated deadline is not necessarily critical to the overall system operation but accumulation of missed deadlines can lead to system degradation. An example of these types of systems are audio applications where occasional missed bits from the audio signal do not necessarily affect the quality of the sound, but a frequent accumulation of missed input bits can lead to lagging in the audio quality, making system degradation apparent to the listener.

Real time systems include two types of tasks: periodic and aperiodic tasks. Periodic tasks routinely execute every period of time while aperiodic tasks only execute at irregular periods of time. An example of a periodic task is sensor data acquisition where a polling function checks input at regular periods of time. However, an example of an aperiodic task include event driven functions such as when a message command is received by the user or a signal is detected in the operating system such as when a segmentation fault occurs. Additionally, all tasks are associated with a priority relaying to the operating system which tasks are the most important to execute. Real time systems are preemptive meaning that high priority tasks can take over processor execution from lower priority tasks, ensuring that high priority tasks can always meet their respective deadlines.

Two common scheduling algorithms for real time systems are rate monotonic and earliest deadline first scheduling. For rate monotonic scheduling, tasks are assigned priorities based off of their associated priorities. With operating systems where a low value is associated with a high priority task, task priorities are calculated by the function $\frac{1}{Period}$. As such, tasks with lower periods will be assigned high priorities ensuring that they will have the ability to preempt tasks with high deadlines and meet their deadlines in time. The second type of real time scheduler is an earliest deadline first scheduling algorithm. This algorithm is again based on the assumption that for reliable system operation, tasks closest to missing their deadline will need to be prioritized for execution, making sure that no task misses deadline. As such, tasks which are closest to missing their deadlines will be executed first in the scheduling priority queue to avoid any delays of other executing tasks that could result in tasks missing deadlines. Any real time scheduling algorithm has to have various qualities that make it useful for reliable and predictable system operation. These qualities include high schedulable utilization, stability under transietn overload, the ability to appropriately schedule in aperiodic tasks, resource scheduling, and low scheduling overhead [75]. These qualities are described below.

- **High Schedulable Utilization** - An algorithm has to have the ability to have a high utilization while feasibly scheduling the set of periodic tasks without missing deadlines.

- **Stability Under Transient Overhead** - Algorithms have to have the ability to handle cases where stochastic execution times can lead to a utilization greader than the schedulable utilization bound.

- **Aperiodic Tasks** - Aperiodic tasks have to be relabile scheduled without missing deadlines while also not effecting the timely execution of routine periodic tasks.

- **Resource Sharing** - Resources have to be effectively shared between tasks while using semaphores and mutexes to establish critical sections.

- **Low Scheduling Overhead** - Algorithms should aim to have as low of a performance overhead as possible on a system.

For a real time CPS system to effectively ensure safe operations, fault tolerant recovery mechanisms have to be put in place to ensure no system crashing, and reliable system functionality. For these mechanisms to be successful, the real time properties of the system need to be maintained, even in the event on having to recover to another process. For example, recovery mechanisms have to guarantee that no hard real time deadlines will be missed while minimizing the amount of missed soft real time deadlines in the system. Additionally, the recovery mechanism needs to ensure a smooth transition in regards to scheduling and that the new scheduler will be able to feasibly schedule the respective tasks for future operation. The two recovery techniques roll forward recovery, and roll backward recovery will be discussed further below.

II.3.2.1 Forward Recovery

Many critical CPS have timing dependability requirements based on the interactions with the surrounding physical environment. As such, any disruptions to these timing requirements due to various circumstances such as system faults or bugs can result in system failure. A fault tolerant system is described as a system that can continue to meet timing constraints with accurate results regardless of the existence of faults [76]. One of the most important aspects of fault tolerance are the detection and recovery stages [77]. Fault tolerant techniques known as roll forward recovery techniques focus on keeping systems up and running normally after a software fault. As such, instead of focusing on the reason for the fault or attack, these systems focus on future operation and how to restore reliable operation as fast as possible. In the event of close approaching deadlines, backward checkpointing techniques can result in a performance overhead leading to missed task deadlines [78]. Three fault tolerant roll forward recovery techniques include redundancy, replication, and diversity.

Critical CPS are considered dependable systems. For a system to be dependable it needs to have fault avoidance, fault tolerance, error removal and error forecasting capabilities [79]. Fault avoidance is how can a fault be prevented from occurring, fault tolerance is the ability of a system to continue functioning in the event of a error or fault, error removal is the ability to minimize the amount of errors in a system, and error forecasting is the ability to predict the exist of system errors in the future. For critical operations in a system, a fault tolerance technique called redundancy can be utilized to guarantee the integrity and availability of the system. Redundancy involves implementing multiple instances of the same component with the intent of comparing results to obtain a consensus on what the correct output should be. For example, if a speed sensor is deployed on a car, if that sensors becomes compromised due to a cyber attack, or misbehaves due to a fault, the vehicle controller will be disrupted by using faulty speed data. However, if 5 instances of the speed sensor are deployed in the car, if one of the sensors becomes compromised, then the other 4 instances will likely still be producing correct output. If a comparison algorithm is implemented, the vehicle controller could then figure out the anomaly in the speed sensor, and switch to relying on the correct output from the other 4 speed sensors in the car. Redundancy can be implemented from both a hardware and software standpoint. In regards to hardware, examples of redundancy include extra transistors, power supplies, sensors, or actuators.

Redundancy is categorized into static and dynamic redundancy groups [80]. Static redundancy involves components used in a system where component failures are masked from, and not perceived by the environment. This means that redundant components are constantly utilized in a system regardless of if a fault is detected, and that output from each component will be received by the rest of the system. An example of this can include having multiple temperature sensors constantly operating in a nuclear powerplant, making sure that regardless of if an error is detected, there will be multiple sensor values to compare against. Dynamic redundancy involves providing an error detection capability within a system which has to be supplemented by redundancy elsewhere. This means that at the start of a program, only one component will be active, but after an error or attack is detected another similar component will be triggered to be activated and serve as a backup version to the original component. Usually static redundancy is more resource intensive compared to dynamic redundancy due to the need for requiring separate components to constantly run regardless of errors [81]. However, static redundancy also provides advantages compared to dynamic redundancy in that no failover time is required and short term reliability is considered to be the highest possible. Furthermore, two types of redundancy implementations include structural and time redundancy. Structural redundancy involves using multiple resources to accomplish a function. One example that can be considered is a packet being sent twice between two points by two different network paths. Time redundancy involves using a longer time period to accomplish a function. An example of this implementation is a packet being sent twice between two locations via the same network path to ensure the message was successfully received.

Both software versions and hardware versions of redundancy have been implemented in critical high availability applications. From a software standpoint, redundancy often involves replicating program code into multiple processes [82]. This provides advantages by decreasing component cost of buying extra processors and sensors. Hardware redundancy implementations exist commonly in the wirless sensor network domain [83]. Additionally, in safety critical applications such as airplanes, flight control computers often have a triplex dynamic redundancy structure allowing for two levels of errors before the pilot has to operate the plane manually [84]. By adding a degree of redundancy to safety critical systems, designers can guarantee constant availability while also including functionality to ensure integrity of system communications.

In distributed systems, processes interact by exchanging messages via various communication links. Process output can either be correct or incorrect. Incorrect process output can lead to propagating errors throughout the rest of the system. Replication and redundancy are very similar in nature. Redundancy involves utilizing multiple instances of the same component with the hope of utilizing another component instance as a backup in the event of a fault. Redundancy often concerns the guarantee of availability in critical applications. Conversely, replication concerns integrity in ensuring that the output is correct. Additionally, redundancy is mostly associated with hardware components, while replication is associated with strictly software components. As such, replication can serve as a cheaper alternative than redundancy due to the reduced cost of purchasing extra hardware components [85]. To accomplish this task, multiple instances of a component will be executed in parallel and the output will be compared to obtain a consensus of what the correct value should be. This is advantageous in situations where one software process is attacked to spoof bad data to a controller, in the hopes of disrupting correct operation. In this instance due to the fact that multiple instances of the same component will be outputting recorded values to the controller, the anomaly value could be detected and the controller can switch to exclusively using the output of the other components, ignoring the output from the attacked component.

There are two types of replication techniques: primary-backup replication and active replication [86]. In primary-backup replication, one process serves as a centralized control while in active replication control is not centralized. In the primary backup replication technique the primary component receives an invocation request from a client process to perform an operation. After the output is computed, the primary component sends an update request to each backup component updates their state and send an acknowledge to the primary. With active replication, each component performs the operation independently and returns the output to the client. The client can either process the input by taking the first response or obtaining a consensus of all of the output values.

Multiple examples of replication exist including N-Version voting, distributed systems, and distributed databases [87]. As such, in each of these systems, each replicated component has the following similar client interaction characteristics: a client process has the ability to send requests to each replicated process until an appropriate reply is received, a client process can send requests to all of the replicated processes and select one output or synthesize the replies it receives, and a client process can pick a replicated process response to use based particular criteria such as speed, or reliability. Additionally, redundancy has been used for cybersecurity applications such as producing secure communication channels, verifying access control, and encryption key management [88].

With respect to cybersecurity, the recovery process needs to limit the ability to successfully attack the system. As such, if an attacker is successfully able to attack a controller before, they should not be able to accomplish this in the future. Diversity is a fault tolerant technique that utilizes moving target concepts to accomplish this task. Diversity is an extension of redundancy and replication in that it is not just enough to include copies of components to prevent system malfunctioning and crashing, but it is also important to make sure that those backup components have a different structure than the primary component so that the system can be as protected as possible against malfunctioning [89]. For example, since each backup component will have a different makeup compared to the primary component, there will additionally be a different vulnerability and fault surface for those backup components. Therefore, if a good diversity approach is taken, if the primary controller fails because of a bug in the software, if the backup controllers are appropriately designed in a diverse way, they will not include the same bugs as the first controller, and therefore not fail due to the same reasons.

Diversity can be grouped into two groups: hardware and software diversity [90]. Hardware diversity is when multiple different versions of hardware such as processors, sensors, and motors are used in the design of a system. Additionally, software diversity is when multiple versions of software components are included in the design such as different processor architectures, different software implementations, and different operating system properties. One of the most common versions of software diversity is N-Version programming. N-Version programming consists of implementing multiple versions of the same program that accomplish the same semantic task, but have a different method for accomplishing that task [91]. For example, these may include different programming languages, a different computation algorithm, or even different function usage. Software diversity can also include data diversity where different types of data representations can reduce the the amount of reconnaissance knowledge an attacker can gain to exploit data formats in a system [92].

Some examples of diversity implemented in the literature include network diversity, and sql database diversity. Network diversity can be utilized to ensure that the probability of successful communication is maximized between two locations, based on utilizing different routes within the network [93]. For example if one communication path becomes inoperable, a communication algorithm can compute a new path of communication, based off of the various alternative network paths between the two locations. Additionally databases incorporate various diversity principles to reduce the probability of a total system crash [94]. To accomplish this, designers can implement multiple database designs in parallel such that when one database fails, all of the databases will not fail.

II.3.2.2 Rollback Recovery

In contrast to roll forward recovery where applications are more concerned about minimizing the recovery time taken to get the system back up and running, some applications need to be concerned about the safety and reliability of system behavior once it recovers. As such, in these cases it is important to keep the system as up to date as possible so that once the recovery process is done, the recovered system will be running with the same behavior as before utilizing the state of information before the fault occurred. Rollback recovery assumes that a system has a memory storage device, where the system state is saved periodically through checkpoints. Once a failure occurs, the system will recover with the last saved checkpoint state, reducing the amount of lost computation conducted before the fault [95]. A lot of research has been conducted on determining what information, what method, and how often a system checkpoint should be saved to ensure recovery process as reliable as possible [96; 97]. The combinations of these checkpointing techniques form the basis of the two most common rollback recovery techniques: state machine control, and recovery blocks.

In the event of a fault tolerant system utilizing redundancy principles, multiple components exist for the purpose of providing backup capabilities in the event of the primary system crashing or being attacked. However, there needs to be predefined mechanisms in place to determine the appropriate recovery decisions once an event is detected. State machines are a useful toll for designing principles appropriate for a safe and reliable recovery plan. A state machine is deterministic modeling approach where different system attributes are defined in a "state" and different events causing changes to states are defined at "state transitions." State machines can be used for the purposes of determining what processes can use a resource at a given time, and analyzing a program flow under various inputs. However, under a recovery scenario, state machines can be used to determine which components to recover to after certain events [98]. The system designer can take into account the various attributes of the redundant components to develop a recovery plan that minimizes the potential risk to the system. For example, in a system with 5 servers where each server is in a different location compared to the first server. In the event of a power failure covering the first three servers, the state machine could insert a state transition to give control one of the last two operating servers. Every state transition in a state machine is dependent on a constraint, specifying different circumstances that have to be true for the state to change. These constraints decide which state machine will be recovered to in the event of the previous situation. One constraint to decide which server to recover to could be which has the lower utilization rate at that point. Another constraint factored in could be to hand over control to the server the farthest distance away, predicting that there will be less of a chance of a power failure in that location.

Additionally, the same approach can applied in software to replication scenarios such as databases on a server [99]. When the first database goes down, the website can recover to utilizing the second or third databases depending on a termination protocol. One example of this protocol could be the client 1 users now being transitioned to using the second database site instead of the first. This additional communication distance may add some lag time to the client 1 users, but the availability will remain. In addition to permanent faults that cause systems to be down until manual intervention is performed, transient faults cause a big risk to CPS by causing portions of a system to be temporarily corrupted without as obvious system consequences observable by the operator or user. One state machine approach was developed to deal with the occurrence of transient faults in a redundant distributed system, while controlling the reconfiguration process of the components [100]. Additionally, state machines have also been utilized for the fault detection, and recovery in optical networks in the past [101].

Recovery blocks are the last fault tolerance recovery mechanism. This technique revolves around the the dynamic redundancy software fault tolerance approach, using three parallel software elements. These software elements consist of primary module which is responsible for executing all of the critical software functions, an acceptance test which tests the primary module output after each iteration, and at least another primary module that can begin to execute once a failure is detected by the acceptance test [102]. Before each recovery block is executed a checkpoint will save the current state of the system, after each acceptance test fails, the checkpoint will be restored, and the alternative primary module will attempt the software function again. This process will be repeated until there are no other alternative primary modules left, or the acceptance test is successful. After the acceptance test is successful the checkpoint will be discarded and the program will proceed.

There are four possible causes that can lead to an acceptance test failure including an error within the executing function, failure to terminate leading to a timeout error, implicit error detection such as protection violations, or divide by zero errors, or explicit error detection which exhaust the recovery capability at that level [103]. Throughout the entire process, a variable exists that stores the recovery level or current alternative primary module version. This illustrates the module that is not only being used currently, but will be used going forward if the acceptance test proves successful. For the purpose of restoring the checkpoint in the event of an acceptance test failure, the system must be able to restore the previous saved checkpoint environment. To accomplish this a recovery cache must be used to store the previous saved variables [104]. By implementing a recovery cache to store only the necessary specified system variables, memory storage can be decreased. Some examples of recovery blocks include an implementation in the sequential pascal language [105], as well as in applications with distributed cooperating process [106].

II.3.3 Simplex Architecture

Simplex based architectures are a common established solution in the area of fault tolerant and resilient systems [107]. Oftentimes, critical CPS require high performance, complex controllers with millions of lines of code. The high complexity of this software leads to it becoming very hard to verify the safety, security, and reliability of the system, resulting in potential exploitable vulnerabilities. Without backup mechanisms this can lead to system faults and crashing which significantly reduces system availability. Simplex aims to solve this problem by creating an architecture that leverages the performance benefits of the complex controller in conjunction with the reliable output of a lower performing, robust safety controller. Execution is primarily handled with the complex controller. However, a decision module exists to monitor the physical state of the system, at which time will transfer execution to the safe controller in the event that unsafe behavior is detected. The Simplex architecture is illustrated in Figure II.8.

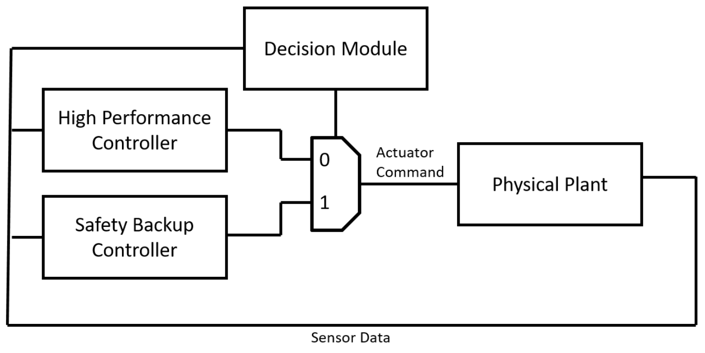


Figure II.8: Simplex Architecture

With the vast established attack surface of automobiles it is important to create security protections that are reliable against known exploits such as code injection attacks, but also robust enough to protect against zero day attacks. As such, a defense in depth approach is required to create many layers of backup security mechanisms to decrease the probability of a successful attack. Moving target defenses such as ISR serve as a valuable solution to this problem by periodically changing the instruction architecture of executing software, voiding the previously gained reconnaissance knowledge of the attacker and making any future payload injection attempt fail due to an invalid instruction exception. However, the larger CPS specific problem that our paper attempts to solve is how can we create a MTD security framework where system operation remains safe, and stable when under cyber attacks. In CPS crashing is unacceptable as any denial of service potentially leads to devastating consequences such as vehicles crashing. As such, our paper focuses on detection, and recovery mechanisms based on the Simplex architecture to safely, and rapidly transition between controllers in the event of cyber-attacks.

The original simplex architecture is comprised of a three component control design with a complex controller, safety controller, and decision module which determines which controller to utilize based on safety and performance concerns [108]. However, there have been several tweaks over the years aimed at optimizing the architecture for various applications. Some notable newer simplex based implementations include Secure System Simplex [107], Net Simplex [109], and L1 Simplex [110]. Net Simplex was developed to optimize the fault tolerance of distributed CPS with networked components, while L1 Simplex presents a fault tolerant system for CPS control system by including stability envelope monitoring in the safety decision module. Secure System Simplex introduces the concept of security to the Simplex architecture by including a side channel monitor integrated with the decision module to optimize the system protections against side channel attacks. Some notable application domains integrating Simplex based control architectures include F16 aircraft flight control systems [111], pacemakers [112], and unmanned aerial vehicles [113].

II.4 Real Time CPS

Real time CPS, especially systems like automobiles face complex design challenges due to the distributed nature of functionality, different manufacturers making different parts with different specifications, and strict timing constraints. As such, these CPS are very hard to integrate, and further schedule tasks in a manner that minimizes missed deadlines. There have been efforts by companies in several industries to smooth the integration process by standardizing specifications for various components. One good example of this is a framework developed by BMW that automates the timing and task scheduling process for automobile design based on the AUTOSAR specification [114]. This concept enhances the ability of designers to think about system timing constraints more accurately, efficiently, and earlier in the design process to enable more predictable system operation at deployment time. Another challenge unique to real time CPS is the tightly coupled nature of the cyber infrastructure with the physical environment. This interaction with physical processes increases the probability of interruptions in the data collection process, receiving inaccurate information which can lead to effects in the computation results. It is noted that three of the biggest real time CPS challenges are maintaining predictable operation with disruptions in the sensor input stream, resolving conflicts between fault tolerance and security concerns, and ensuring the design correctness of the system [115].

It is additionally noted that the controller area network (CAN) protocol has several security flaws that make it vulnerable to attack. When looking at the design of CAN based communication systems, there are two layers of security to focus on: the internal CAN network and the external applications providing access to the internal network. CAN provides a limited bus bandwidth and message length which presents a challenge for implementing message authentication [116]. This design opens up vulnerabilities to masquerade and replay attacks which take advantage of the lack of authentication [116]. CAN based systems, which are often times real time systems in nature could have violations of design constraints due to impedances in system performance caused by the overhead of security mechanism implementations [117]. Additionally, in the case of automotive systems, there are dozens of electronic control units (ECUs) connected to the CAN network to provide functional operation [118]. The CAN network is utilized for these complex interconnected systems due to the reduction in necessary connecting cables because of a common connection to the bus as well as simplicity of packet and communication structure [119]. However, because of the widely shared hardware/-software platforms between vehicle manufacturers, the CAN packet identifier fields are consistent across vehicle models allowing attackers to execute spoofing and denial of service attacks by injecting packets on the network [119]. Furthermore, since the CAN protocol broadcasts messages throughout the whole network, the ability to access the internal network allows the attacker to spoof messages to any ECU in the vehicle, potentially causing problems with safety critical systems [120]. In [121] the researchers demonstrated that an attacker could control the windows, dashboard warning lights, airbag control system and gateway ECU by using the above techniques and in [122] the power steering module was stalled completely.

When introducing moving target defense techniques such as ISR, ASR, or DSR into existing real time software implementations, performance overhead is created that wasn't taken into account during the initial design of the system. As such, with the added performance overhead of security mechanisms, controller execution times can increase to the point of exceeding their previously assigned deadlines. In this case, it is imperative that the system be dynamically readjusted to adapt the associated sampling rates of underlying controllers. Generally, as the sampling rate decreases, the input-feedforward passivity index of the system decreases. The input-feedforward passivity index is a characterization of the minimum phase behavior of the system, which is most often associated with delay. As sampling rate decreases, the system is unable to react to changes in the system. This behavior is characterized using the input-feedforward passivity index, in which a negative index value indicates non-minimum phase behavior. In order to ensure stability and safety of the closed loop system for adaptive sampling rates, passivation methods described by input-output transformations that generalize typical methods of series, feedback and parallel (or feedforward) interconnections to passivate a system need to be considered [123]. The range of appropriate sampling rates need to be defined at design time to establish a safe bounds of operation for the system.

One important challenges to consider with regards to real time control reconfiguration is the missing of deadlines during the reconfiguration process, and the resulting effect on controller behavior. Many control algorithms rely on a steady rate of sensor input data to be able to maintain passivity, and stability of the physical dynamics. However, with the absence of input data, the controller can verge into an unsafe state, resulting in potential physical damage. With regards to the reconfiguration process, there is the possibility that there will be a loss of feedback data due to the reconfiguration process. It is possible that attack detection and reconfiguration cannot be completed within a single sampling period, preventing the controller receipt of sensor data.. This situation is similar to packet loss in networked control systems and has been addressed by developing passive sample and hold components that guarantee passivity in the presence of information loss and time-varying delay [124; 125]. This approach further establishes that it is possible for recovery to occur that preserves passivity and safety in the instance of an attack occurring within a sampling period. Furthermore, it has been shown that a sustainable solution is to specify reconfiguration policies in advance to preserve controller passivity under various scenarios of recovery.

The Simplex architecture, which is what our developed reconfiguration architecture is based off of, aims to create a fault tolerant system by implementing redundancy to execute various types of controllers in a system. At its most basic level, Simplex includes an high performance controller, a safe state controller, and a decision module. The high performance controller executes by default, and when a fault occurs such as a missed deadline, or invalid output, the decision module will transfer to execution to the safe state controller which is less efficient, but is more reliable, and predictable. In most versions of the Simplex architecture, all three of the components are located at the application layer. However, it has been noted that when operating Simplex on Real time operating systems, internal bugs in the operating system can be left unchecked by the decision module resulting in system crashing, a consequence in which Simplex was designed to prevent. The developed architecture System-Level Simplex aims to solve this problem by providing a means for providing hardware/software partitioning by moving safety critical components such as the decision module, and safety controller to isolated processing unit [112]. Furthermore, another approach introduced for ensuring a smooth transition in Simplex recovery is the unification of linear matrix inequality, and hybrid systems reachability analysis within the decision module [126]. Instead of utilizing these techniques independently, this approach allows for more accurate decisions by the decision module, enabling more reliable use of the complex controller, versus the significant use of the less optimal safety controller.

To ensure proper real time operation of CPS, run time assurance monitoring is necessary. This involves the checking of complicated properties that involves analyzing many system variables. Thus, this procedure produces a significant overhead to the system. To reduce this overhead, a lot of literature work has focused on combining static analysis techniques with dynamic analysis runtime checking. Additionally, symbolic techniques in the compiler optimization process have been explored. The main tradeoff that must be taken into account is between monitoring accuracy, and system overhead [127]. The designer must take into account whether it is acceptable to miss occasional execution events that potentially could effect accuracy, or check every execution event for maximum accuracy, but consequently increase the probability of missing a deadline due to the higher overhead. Simplex based architectures rely on recovery mechanisms that make the assumption that the fail safe controller will be effective in all scenarios once a violation is detected by the monitoring module. To solve this problem, developed architectures have focused on incorporating diagnostic capabilities, and user specified recovery plans to determine the appropriate path to take once a violation is detected [128; 129; 130]. Another approach taken instead of recovery is called runtime enforcement which attempts to prevent potential violations by delaying and reordering events leading to various unsafe states. Runtime enforcement architectures have been proven to be effective for security applications, but relies on the ability to block events to allow for real time checking [131; 132; 133]. Runtime enforcement produces a higher overhead compared to a strictly recovery based system under normal conditions, but produces a higher level of security for the specified property violation. A proper balance between security, and overhead can be established by utilizing runtime enforcement for the most critical security violations, while relying on recovery for unexpected less critical violations.

CPS designs must take into account reliability, physical, and timing constraints. From a real time perspective, timing constraints are the most significant aspect. In regards to the Simplex architecture, several timing considerations are noted including the complex controller computational complexity, the monitoring module complexity, the switching logic complexity, hardware nondeterminism, timing effects from stochastic faults, and timing effects from the software implementations concerning control flow, memory management and algorithm development [134]. Scheduling algorithms, worst case execution time, networking, and faults all play a role in these timing considerations. In CPS, which are often times hard real time systems, the three most common scheduling algorithms are round robin, fixed priority preemptive scheduling, and rate monotonic scheduling [135]. To accurately utilize these algorithms, appropriate deadlines need to be established based on determining the worst case execution time of a task. The worst case execution time can be established by either producing a hard bounds of performance based on rigorous testing, or an estimation of the bounds based on static analysis of a programs execution [136]. All real time implementations over approximate the worst case execution time of a task to ensure that the program always completes before the respective deadline time constraint occurs. Due to the distributed nature of Simplex tasks as well as CPS systems in general, synchronization algorithms are necessary to establish a global clock for communication between components [137]. Finally, faults such as Byzantine failures, and hardware faults can lead to cascading timing issues in a system [138]. Real time assurance efforts have been underway to help address these timing considerations. For verifying that time constraints are satisfied at runtime, many approaches utilize timestamps and clock synchronization [139]. Additionally, metric based temporal logic has been used to specify quantitative properties to monitor [140].

Three types of approaches have been utilized to address the performance overhead created from real time monitoring and recovery techniques. These approaches include time triggered monitoring, networked monitoring, and probabilistic time aware monitoring. Time triggered monitoring involves tasks being initiated at specific periods in time, versus an external event occurring [141]. As such, instead of relying on generated event signals monitoring tasks are based on periodically sampling system state variables. This approach is effective in reducing performance overhead and has been utilized in various safety-critical applications [142]. However, this approach requires extensive analysis to determine appropriate sampling periods for tasks. A sampling period too low will not be able to be efficiently utilized with a rapid detection time, and a sampling period too high will take resources away from other system tasks. The second approach, networked monitoring, implements the monitoring module of the architecture on a separate external piece of hardware, ensuring that no overhead will be presented to the currently executing controller [143]. This approach is the most optimal from a performance standpoint, but requires a higher cost due to requiring two separate pieces of dedicated of hardware, as well as external communication is required for recovery instead of the more direct interface available when operated on the same piece of hardware. The last approach is probabilistic time aware monitoring. This approach relies on computing a probability of a property violation based on output from a learned Hidden Markov Model given an input trace of system activity [144]. This approach is beneficial in determining the hidden state of a system at a time instance not currently sampled. As such, the sampling rate of a monitoring system can be decreased, consequently reducing the associated performance overhead on the system.

II.5 Mixed Time and Event Triggered CPS

In safety critical CPS, there is often a tradeoff between predictability and flexibility. Advances in computing technology have allowed for a more digitized environment in safety-critical systems such as automobiles, aircraft, and medical devices. With this rapid transformation, it is important to ensure that the critical predictability properties remain in tact, as any failure in correct system behavior can result in significant damage to the surrounding environment. As such, in CPS each computation task must be completed by a predetermined deadline in order to ensure that each actuation event executes at a rapid enough frequency to support safe operation. There are two types of real time systems including hard real time and soft real time. Hard real time systems mandate that all system deadlins must be met. Any failure to meet this requirement is considered a failure in the system. As such, systems must ensure a high level of predictability and go through rigorous verification processes to ensure no corner cases before they are deployed to commercial use. The second type of real time system is a soft real time system in which the goal is to not miss any predefined system deadlines, but it is acceptable to miss an occasional deadline without causing system failure. Generally, even though system failure will not occur when a deadline miss occurs, an accumulation of deadline misses has the potential for causing system performance degradation.

There are two types of scheduling approaches utilized to support hard and soft real time properties in safety-critical CPS: event triggered and time triggered scheduling. In event triggered architectures, the system activities, such as sending a message or starting computational activities, are triggered by the occurrence of events in the environment while in time triggered architectures activities are triggered by a sequence of global time [145]. As such, event triggered architectures are controlled through interactions with stimuli in the environment, while time triggered architectures are autonomously controlled through a static predefined schedule of events, supporting autonomous control flow. The advantage to time triggered architectures is that at any given point of time, the system user will always be confident on what process is executing, providing a degree of predictability in the system operation behavior. As such, systems can be fully verified with 100% certainty that no corner cases will occur resulting in system failures. This makes time triggered architectures the approach of choice for hard real time safety-critical systems that necessitate a high degree of reliability [146; 147]. For soft real time systems in which safety is not a direct result of predictable operation, event triggered architectures are preferred due to the higher degree of flexibility and more efficient performance.

Systems can be designed with dynamic resource allocation and resource sharing strategies geared to the average use case, meaning that only during worst case outlier events will timing property violations occur. This makes system development less expensive and resource intensive, decreasing the time to market.

In the past, these types of architectures were generally considered mutually exclusive. However, recent research has focused on combining the predictability and reliability of time triggered architectures with the flexibility of event triggered architectures to support optimal operation of safety critical CPS [148]. This hybrid approach utilizes a time triggered periodic schedule for normal system operation, while event triggered sporadic events are reserved for less frequent event occurrences such as error detection or an aperiodic communication event. By building in enough slack into the static schedule at design time, these systems can be verified to be fully schedulable, meaning that at any point a sporadic process can execute without disrupting the timing requirements of the time triggered tasks. One of the most popular implementations of this approach is the ARINC 653 standard which is commonly found in aviation, automobile, and space designs [149; 150; 151]. In our approach we utilize a component based ARINC 653 implementation to support time triggered and event triggered capabilities in our architecture [152].

II.6 Comparison with Proposed Work

As CPS continue to increase in complexity and scale, cybersecurity becomes more and more of a challenge. The attack surface is constantly changing with new avenues being created for attackers to compromise a system. Additionally, the CPS domain is unique in that due to the tightly couple nature of CPS with the physical environment, once a device is compromised, an attacker can not only retrieve data like in traditional information technology applications, but can leverage actuation capabilities to inflict massive damage to the surrounding areas. With the added need for always guaranteeing system availability, especially in critical infrastructure, it is an even bigger challenge to recover from cyber-attacks in a manner that keeps systems up and running in a safe and reliable manner.

CHAPTER III

Integrated Instruction Set Randomization, Address Space Randomization and Control Reconfiguration for Securing CPS

III.1 Introduction

With the increasingly connected nature of Cyber-Physical Systems (CPS), new attack vectors are emerging. Normally, an adversary will use memory corruption attacks to achieve manipulation of the cyber sub-system, leading to alteration of the physical dynamics. As such, the compromise of safety-critical systems, as well as commercial Internet of Things (IoT) devices opens the gates for attackers to exfiltrate sensitive data, or inappropriately control actuation. It is critical to shift the CPS security focus into a more proactive approach, aimed at creating more resilient architectures.

Automobiles today are extremely complex systems of systems, consisting of several hundred electronic digital components with over a million lines of code. The internal automotive network consists of a series of multiple communication buses such as CAN, LIN, FlexRay, and MOST [27]. Due to the traditionally standalone design of vehicle architectures, the communication and controller designs prioritize functionality and cost over cybersecurity. Additionally, with the majority of software being written in legacy code, vast numbers of vulnerabilities are potentially included. With the introduction of external interfaces such as infotainment centers and telematics systems, adversaries now have remote avenues in place to access the internal vehicle network [7].

The two primary instances of memory corruption attacks are code injection and code reuse attacks. Code injection attacks exploit existing input vulnerabilities for injecting a custom designed instruction payload that can be executed by control flow redirection [153]. For code injection attacks to be successful, the adversary has to rely on knowing the native instruction set architecture of the target machine. Code reuse attacks on the other hand leverage existing code by diverting control flow to legitimate code segments allowing the adversary to achieve his/her malicious goal even in the cases where directly injecting code is not possible [154]. One of the most popular examples of this type of attack is return oriented programming (ROP) [155] in which case existing code gadgets are chained together to form a program that can execute malicious behavior. One of the most common memory corruption vulnerabilities in legacy code leading to code injection and code reuse attacks is the the buffer overflow. Buffer overflow vulnerabilities allow attackers to input data longer than designed, overflowing into adjacent areas, and if properly designed, can be leveraged to redirect control flow.

Moving Target Defense (MTD) aims to prevent legacy vulnerabilities by dynamically changing system properties. Compared to traditional defense mechanisms which focus on identifying malware, and suspicious communications, MTD focus on decreasing the reconnaissance knowledge of the adversary with the goal of minimizing the probability of successful reverse engineering, vulnerability discovery, and exploit deployment. Two MTD techniques utilized in this paper are Instruction Set Randomization (ISR), and Address Space Randomization (ASR). ISR is a technique for protecting against code injection attacks by changing the binary instruction set architecture to a randomized version that is not known [156]. ASR is a technique for mainly protecting against code reuse attacks by introducing diversity in the various segments of a program to make external memory access unpredictable. ASR can be implemented at various granularities including course grained [157], and fine grained [58], while also having the ability to be customized to protect the most critical memory segments [59].

In the CPS domain, even when successfully protecting against cyber-attacks, it is equally as important to maintain reliable, safe, and predictable operation of the system. With ISR and ASR deployed, code injection and code reuse attacks will be thwarted, but an invalid instruction or invalid address access exception will be generated, leading to program termination. In this sense, it is not acceptable for a safety-critical system to stop functioning, as any loss of availability can lead to unsafe actuation causing physical damage. As such, there has to be recovery mechanisms in place to keep the system up and running at all times, even when under a cyber-attack campaign.

To address the difficulty of guaranteeing system availability, while preventing code injection and code reuse attacks, we have developed a security architecture that includes an AES 256 ISR implementation for protecting against code injection attacks [158], combined with a fine grained ASR implementation for protecting against the relative, and direct control flow redirection necessary for code reuse attacks [3]. Our security architecture consists of three stages including attack protection (randomize, derandomize), detection, and recovery. The main CPS challenge addressed in this chapter is protecting system integrity during cyber-attacks, while maintaining system availability with safe and reliable operation. This chapter makes the following contributions:

- We develop a CPS security architecture for providing secure protections against code injection and code reuse attacks by utilizing AES 256 ISR, and function level fine grained ASR.
- We incorporate control reconfiguration into our security architecture for maintaining system availability in the event of a cyber-attack.
- We implement a hardware in the loop testbed prototype using a combination of off-the-shelf embedded computing hardware and open source simulation software for analyzing the effects of cyber-attacks and our security architecture in CPS environments consistent with deployment settings.

- We present an autonomous vehicle case study to demonstrate the effectiveness of our security architecture in limiting the physical impact of code injection, and code reuse attacks on driving safety.

III.2 System Model

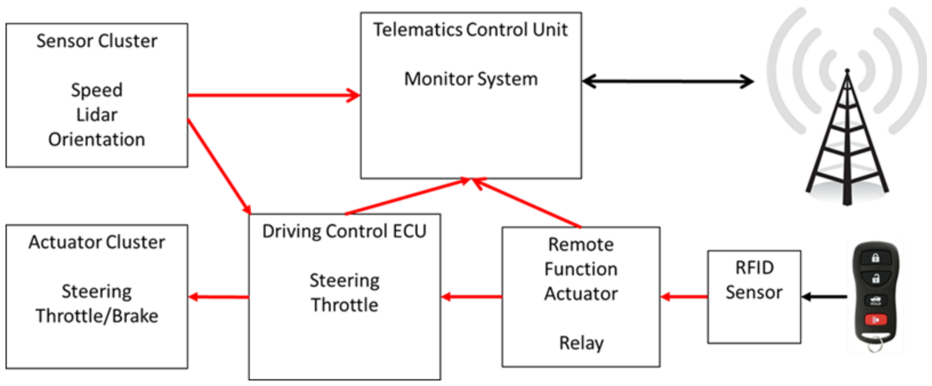


Figure III.1: Vehicle Architecture Diagram

An exemplary vehicle system model is shown in Figure III.1. This model includes 6 components: a sensor cluster, actuator cluster, driving controller, telematics control unit (TCU), remote function actuator (RFA), and RFID sensor. The sensor cluster provides critical data representing the current state of the vehicle such as the speed, position on the track, and heading. The actuator cluster provides the ability to manipulate vehicular behavior such as steering and acceleration. The driving controller is responsible for performing computation based on the provided sensor cluster input, and outputing commands to the actuation cluster. Both the TCU, and RFA are responsible for providing the external interface for the vehicle. The TCU monitors the various metrics of the system, transmitting data to a remote operating station for maintenance and emergency purposes. The RFA is responsible for determining the presence of a key fob for allowing the vehicle to be turned on.

In the system model, the sensor cluster, actuator cluster, and driving controller are on a safety-critical CAN bus network, including both communication authentication to prevent spoofing, and integrity checking within the driving controller to ensure that utilized sensor data is accurate. On the other hand, the TCU, and RFA communicate with the driving controller through a low priority CAN bus interface. Since these components are the most vulnerable to remote attacks due to being connected to external communication channels, the safety-critical and low priority communication buses protect against the TCU and RFA directly controlling the sensor or actuator ECU clusters. However, to detect the presence of the key fob, the driving controller constantly polls for status updates from the RFA. This communication is authenticated to prevent message spoofing, but there is a buffer overflow vulnerability in the driving controller that provides an opportunity for memory corruption attacks.

III.2.1 Attack Model

The attack model for this paper focuses on code injection and code reuse attacks on a vehicle network. The authors in [159] note that the biggest current threat to self driving vehicles is exploitation through remote avenues. As such, the attack vector utilized in this paper consists of the adversary compromising the TCU through the remote cellular interface, and consequently pivoting to hijack the RFA. With access to a direct communication channel with the driving controller, the adversary can craft a message payload to take advantage of the buffer overflow vulnerability and alter control. At this point two options are presented: a code injection attack which inputs executable code directly on the driving controller stack, and a code reuse attack which strategically diverts the driving controller control flow to other locations in program memory. By utilizing these two attack techniques, the physical dynamics of the vehicle can be significantly altered consequently compromising safety.

III.2.2 Problem Formulation

With the possibility of a code injection or code reuse attack on the vehicle network, data integrity is not just threatened but safety can be compromised. In the case of a safety-critical CPS such as an automobile, alteration to normal controller functionality can lead to physical damage. Additionally, a loss of availability, even in the event of successful cyber-attack mitigation can be just as detrimental to the physical safety of the system. The problem that we aim to solve is how to protect against code injection and code reuse attacks effectively, while reconfiguring fast enough to maintain safety and stability of the CPS. We hypothesize that by utilizing ISR and ASR in combination with control reconfiguration within a developed security architecture, we can not only protect against code injection and code reuse attacks, but can maintain safe operation throughout cyber-attack events.

Five assumptions are made for our approach to be successful. First, it is assumed that the sensor and actuator clusters are fully secure. The driving controller ECU contains the buffer overflow vulnerability utilized for control hijacking, while the TCU and RFA contain vulnerabilities allowing for key fob message spoofing. Second, the attacker has full knowledge of the system architecture necessary to craft an accurate payload. Third, the attacker has complete knowledge of the architecture of safety-critical controllers like the steering controller. Fourth, the attacker has knowledge of the beginning address of the buffer input on the driving controller stack. Fifth, the attacker has knowledge of the relative memory location of the current driving controller function return address on the stack from the beginning of the input buffer. After this knowledge is gained, the attacker crafts an input payload to overwrite the current return address to divert control flow to either the injected payload, or existing control function. At this point, the adversary can cause the vehicle to enter an unsafe state by altering the physical behavior of the car. These assumptions are not impractical given examples demonstrated in the literature [31].

In the rest of this chapter we discuss a developed security architecture aimed at preventing the vulnerabilities discussed in our attack model. The objectives of our security architecture include the following:

1. Any implemented software must maintain safe and reliable performance of the CPS. This includes minimizing the security architecture overhead, and ensuring that all real time deadlines are met.

2. Implement reliable detection mechanisms for monitoring and flagging attack events.

3. Implement reliable recovery and control reconfiguration mechanisms to maintain safe system operation and minimize system downtime. This is especially crucial in CPS applications where the cyber controller crashing, even when experiencing a cyber-attack can result in devastating consequences.

To evaluate the effectiveness of our architecture within the context of an autonomous vehicle case study, we utilize a developed hardware-in-the-loop testbed. We further utilize physical metrics such as vehicle position combined with software metrics like performance overhead and recovery time to assess safety in both normal operation and attack scenarios. Finally, to conclude that our hypothesis is true two observations need to be clear from the results: 1) The performance overhead needs to be minimal enough to ensure that execution times do not exceed designed real time constraints and 2) Vehicles need to follow safe driving behavior, maintaining a safe position near the center of the road while avoiding driving off the road or colliding with obstacles. In the event that both of these observations are true, we can conclude that our architecture is successful.

III.3 Architecture

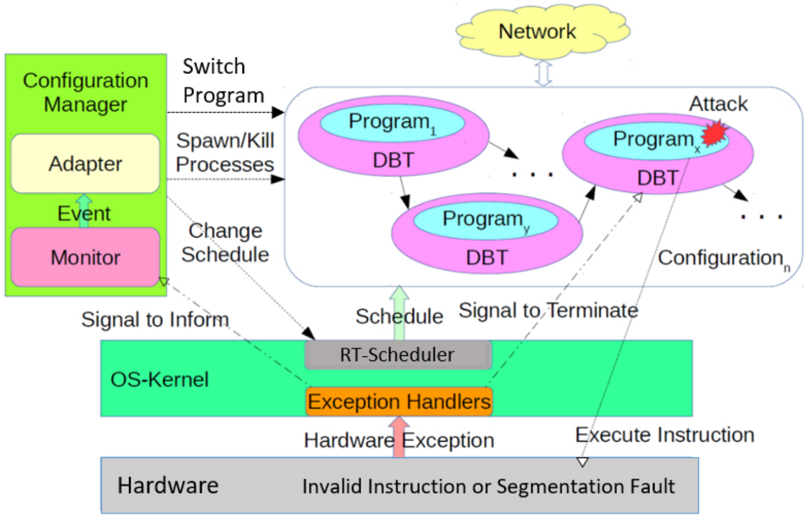


Figure III.2: Control Architecture [3]

In Figure III.2, a high level overview of our security architecture is presented. The key components are the (1) Configuration Manager (CM) that oversees, customizes, and adjusts the operation of the various operating components, (2) CPS Controllers which control the physical plant, (3) Dynamic Binary Translator (DBT) which provides a sandboxed runtime environment for each CPS controller, and (4) Operating System Kernel which handles the task scheduling and exception detection. We assume that each CPS controller in our architecture may be vulnerable to cyber-attacks by the adversary, but the remaining components are secure. Our security architecture is designed with the goal of keeping the CPS controller from becoming compromised by the attacker. These components are described below.

Configuration Manager (CM): This process oversees and maintains the operation of the security architecture, including all underlying components such as DBT, CPS controllers, and network communication. Additionally, the CM is responsible for detecting cyber-attacks, and executing the reconfiguration process to transfer execution to the backup controller in the case that the default controller is compromised. Signal handlers are implemented to capture exception events caused by failed cyber-attacks. After attack detection, reconfiguration algorithms determine the appropriate controller process to transfer to, and execution can be established through the use of POSIX signals.

CPS Controller: The CPS controller controls the physical dynamics of the system through receiving sensor data as input, and outputting actuator commands through the use of computation algorithms. Our architecture allows for incorporating domain specific controllers representative of various CPS applications. The CPS controller is the customized component in the architecture, potentially containing vulnerabilities that can be exploited to achieve code injection and/or code reuse attacks.

Dynamic Binary Translator (DBT): The DBT is responsible for establishing an unique runtime environment for each CPS controller in the architecture. This component manages the customized runtime environment for each controller by initializing a randomization key, randomizing the instruction and address space, and derandomizing instructions as they are fetched at runtime. This component allows for the dynamic generation of randomization keys at load time, ensuring security is maximized by generating different randomization keys for each controller. For our architecture, both AES 256 ISR, and fine grained ASR are supported. This component effectively sandboxes the underlying CPS controller, leaving a code injection or code reuse attack ineffective due to incorrect reconnaissance knowledge. The DBT is additionally responsible for storing the generated randomization keys, allowing for the maintenance of key confidentiality throughout the program.

Operating System Kernel: The operating system manages the scheduling for our architecture, utilizing a rate monotonic scheduling algorithm. Additionally, our detection algorithms in the CM are based on exceptions such as an invalid instruction execution or invalid address access caused by a failed attack attempt. The operating system is POSIX-compliant, enabling signals used for one way communication between architecture components.

III.4 System Implementation

For our security architecture implementation, we focus on a two stage approach, MTD and control reconfiguration. MTD is focused on providing protection against code injection, and code reuse attacks, while control reconfiguration is focused on maintaining system availability in the event of a cyber-attack. These stages are discussed below. The main contribution of our architecture is the integration between these two stages, linking MTD within the DBT to the control reconfiguration defined in the Configuration Manager.

III.4.1 Moving Target Defense Implementation

III.4.1.1 Instruction Space Randomization

To perform a successful code injection attack, an adversary must have knowledge of the system instruction set architecture to craft a valid payload [158]. The adversary will be able to successfully execute code directly on the target system only if the instructions can be validly decoded. However, if the instruction set architecture is not known, the attack will result in the process terminating due to executing an invalid instruction. ISR leverages this adversary requirement by dynamically changing the binary representation of instructions, and decreasing the likelihood that the correct format will be utilized in a code injection attack. As such, the adversary will end up using an invalid instruction representation resulting in control system termination. Our implementation supports both XOR and AES 256 encryption. AES 256 encryption presents a higher overhead to the system, but also provides a higher level of security for safety-critical applications.

Algorithm 1 Instruction Set Randomization Algorithm

```
P = Program Code Segments
/* At Load Time */
key = generateAESKey()
for Instruction I in P do
  | E = AESEncrypt(I , key)
end
/* At Run Time */
N = Instructions Sent To Processor
for E in Fetched Instructions do
  | I = AESDecrypt(E , key)
end
```

At load time we first dynamically generate a randomization key. As the program is loaded into the DBT application memory, an algorithm will encrypt each instruction with the generated key. At runtime, as each instruction is fetched by the DBT, it will first be decrypted with the same key before it is passed along to the processor. In this case, since the instructions are encrypted before runtime, even if the attacker is able to inject malicious instructions into the program with the original format, once the instructions are fetched by the DBT, they will be "decrypted," resulting in a new invalid instruction representation. As such, once these attacker instructions reach the processor, they will result in an exception. The exception will then be detected by the Configuration Manager which then triggers the reconfiguration process in the architecture. The ISR process is described in Algorithm 1.

III.4.1.2 Address Space Randomization

To perform a successful code reuse attack, an adversary must have knowledge of the memory layout, specifically the locations of safety-critical functions [160]. In a normal attack, control flow will be redirected to these target functions to manipulate safety-critical operations even in the case where code cannot be injected directly. Since the target code already exists in the program, ISR is not a feasible protection since that code will already be randomized at load time along with the rest of the program. However, ASR leverages the requirement of knowing the target function location by changing the memory locations of various entities within the program. Since the location of a target function will never be the same for two separate program executions, the adversary will hardly be successful in redirecting control flow to the respective function. There are multiple levels of ASR, the most popular of which is randomizing the base addresses of shared libraries, the stack, and heap sections (e.g. Linux ASLR [60]). However, for higher security applications, our architecture includes a fine grained ASR implementation, randomizing memory locations at function level granularity. There will be a higher level of performance overhead compared to traditional ASR implementations, but this overhead will mostly be limited to load time.

Algorithm 2 ASR Implementation

```
P = Program
F = Function Symbols
for Symbol S in P do
    if S=function then
        F.append(S)
end
for Symbol S in F do
    R = selectRandom(F)
    swapLocation(F,S)
end
for J in Every Jump do
    S = findAssociatedFunctionSymbol(J)
    updateJump(S new address)
end
```

For our implementation, at load time we first iterate through the binary ELF file to find the memory location of all function symbols within the program, storing them in a table. We then iterate through this table, switching function memory positions as we go along. As instructions are fetched by the DBT, branch/calls with absolute addresses are easily handled by patching the respective branch/call instruction with the updated target address. However, indirect branch/call instructions are more challenging to handle. In this case, the target address needs to be accessed dynamically before branch/call patching can occur. To accomplish this, DBT control flow is altered to separate the current basic block into two segments: one consisting of instructions up to the branch/call instruction, and one consisting of the specific branch/call instruction. The first basic block segment is executed including an added instruction to access the register storing the respective target address. At this point, the normal patching process can be executed by checking the target address against the function table, and updating the respective branch/call instruction with the new address. Since the ASR process is completed dynamically for every CPS controller binary at load time, running a program two times will result in not only different memory locations of functions, but also different function orders for protecting against relative jumping. The ASR process is described in Algorithm 2.

III.4.2 Control Reconfiguration

In our security architecture, the binary is randomized at load time, and derandomized instruction by instruction before they reach the processor for execution. Our architecture supports multiple CPS controller instances, but by default includes a default controller, and backup controller configuration. Each of these controllers is sandboxed in a MTD environment within a DBT, enabling both AES 256 ISR, and fine grained ASR with function level granularity. This DBT provides the ability to dynamically customize executing binaries, allowing for us to tap into the virtual pipeline to change memory at load time, and derandomize instructions as they are fetched. The MAMBO DBM environment has been utilized to serve as the DBT in our framework [56].

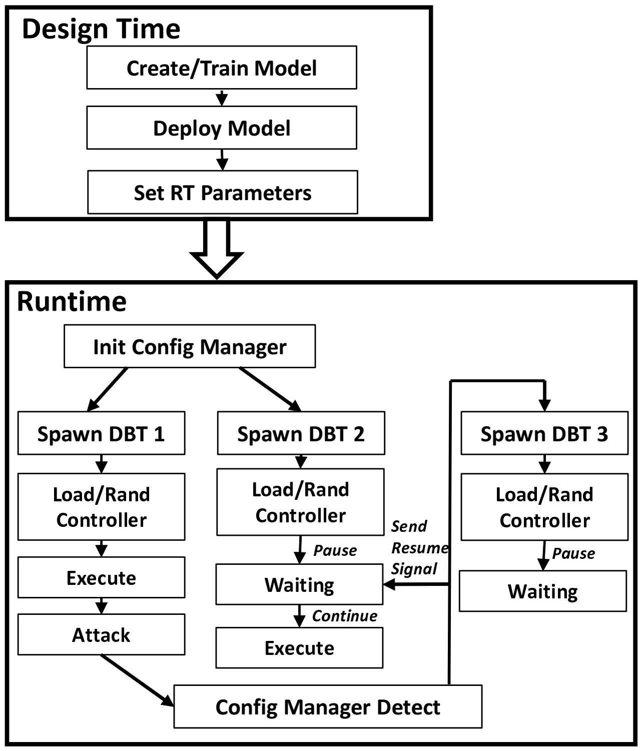


Figure III.3: Architecture Process Flow

When the architecture is started, the first component initiated is the Configuration Manager. The Configuration Manager spawns the CPS controllers inside of DBTs as child processes. This allows the Configuration Manager to monitor the underlying vulnerable controllers for cyber-attacks, as well as any other unsafe behavior. Further, the Configuration Manager controls the execution of the controllers, allowing for the transfer of control in the case of an attack. By default, controllers are built to be put in a waiting state once loaded, and the Configuration Manager then resumes the default controller with a SIGCONTINUE POSIX signal. In both DBT processes, a randomization key is dynamically generated, ensuring that there will be a different randomization key for every component instance. This key is stored inside of the DBT enclosure and is utilized for the derandomization process. Since both controllers are loaded inside of their respective DBT (MAMBO) application memory, the DBT has the full ability to execute the derandomization throughout runtime.

When looking at a snapshot of our architecture process flow, the default CPS controller will be operating under normal circumstances inside of a DBT. The backup controller will exist in a waiting state. As each instruction from the default controller is fetched by the DBT, it will be derandomized utilizing an AES decrypt operation with the respective randomization key. At this point, the instruction will be stored in a basic block data structure and sent to the processor for executiond. Once an attack is encountered, the Configuration Manager has attack detection algorithms that handle exceptions. After this point, the default controller is compromised, and the Configuration Manager triggers the recovery process by transferring execution to the backup controller with a SIGCONTINUE POSIX signal. Afterwards, a new default CPS controller is spawned inside of a DBT enclosure to serve as the new backup controller. By reconfiguring in this manner, a safe state can be ensured during unstable circumstances, while the benefits of the default high performance controller can be maintained during normal operation.

For architecture implementation to be successful two assumptions must be true. The first assumption is that the operating system, as well as the Configuration Manager process are secure. The vulnerable component that we focus on in our threat model is the the CPS controller. The second assumption is that the communication between the Configuration Manager and the DBT processes must be unidirectional. As such, the Configuration Manager will be able to communicate to DBT processes through POSIX signals. By not allowing communication in the other direction the threat of the Configuration Manager becoming compromised through the CPS controller is eliminated.

III.4.3 Recovery Time Analysis

During the course of an attack, it is important to ensure that the CPS maintains safe and reliable operation. As such, it is important to minimize the recovery time as much as possible to maximize normal operation. The recovery process is comprised of three stages: detection, backup controller execution transfer, and backup controller execution. The recovery time noted in this paper is measured from the time of attack occurrence to the time an actuation command is sent from the backup controller.

The first phase, detection, consists of the Configuration Manager determining the presence of an attack. During a code injection or code reuse attack, the consequences will result in an exception which will be caught by signal handler functionality within the Configuration Manager component. This process is handled by the operating system and can be considered negligible in comparison to the overall recovery time. Once the attack is detected through the Configuration Manager, the second phase consists of the process of transferring execution to the backup controller. Since in our implementation, the backup controller is loaded into memory with a waiting state, the Configuration Manager only needs to send a SIGCONTINUE POSIX signal to the backup controller to trigger the process to resume execution. Since POSIX signals are handled by the operating system, the time of this phase can also be considered negligible. The final phase, which encompasses the largest portion of the recovery process is the backup controller execution to compute a new actuation value. With the assumption that the default and backup controller both have the same defined period *P*, the recovery time taken from resuming execution to actuation transmission will be *P*.

During runtime, an attack can occur at any point throughout the period. At best, the attack will occur right after a deadline, allowing the backup controller to produce an actuation command at time *P* later, just after the next deadline instance. However, at worse case an attack will occur just before a deadline occurrence. In this case, the backup controller will take over execution and produce a new actuation command at time *P* later. Since the new deadline will be defined as time *P* after the deadline following attack, the actuation will be successfully computer before the new deadline is encountered. As such, in the worst case scenario, only the deadline immediately following the attack will be missed, meaning that in our approach, only 1 deadline will be missed at worst. By limiting the recovery downtime to one missed deadline, we can resume normal operation fast enough in order to maintain the stability of the CPS.

III.5 Evaluation 1: TORCS Vehicle Platoon

III.5.1 Experimental Testbed

For analyzing our security architecture for CPS, it is important to analyze both the cyber and physical dynamic effects. To maximize the compatibility of the framework, the software must be testbed on platforms consistent with the deployment environment. To support this work, a hardware-in-the-loop testbed was developed for aiding in measuring, and analyzing the cyber-attack effects as well as our security architecture performance overhead. We utilize this testbed for implementing security experiments for evaluating our MTD framework under varying scenarios.

III.5.1.1 Hardware Architecture

Autonomous vehicles consist of a variety of interacting distributed components. As such, the backbone of our testbed revolves around open source embedded hardware. We break up a CPS into various components including the simulation workstation (physical plant), sensors and actuators, and computational components. A local network provides communication capabilities within the distributed CPS environment. To support realistic automotive designs, the local network consists of both a 100 Mbps Ethernet network, and a 1 Mbps CAN Bus network. For implementing high complexity controllers, a NVIDIA Jetson TX2 board [161] is included as the computational platform consisting of a Quad Arm A57 CPU with 256 NVIDIA Pascall CUDA cores. For representing the lower complexity intermediary sensor and actuator software in the ECU cluster, Beaglebone Black 1 GHz ARM Cortex-A8 embedded computing boards [162] are included. Finally, the simulation workstation consists of a single i7 desktop computer with a 7200 RPM hard drive. This hardware setup is illustrated in Figure III.4.

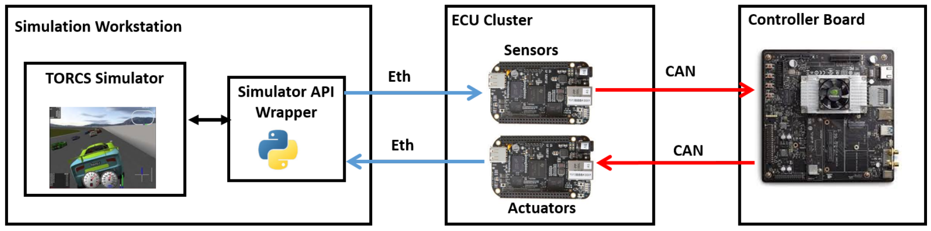


Figure III.4: Testbed Hardware Architecture

III.5.1.2 Software Architecture

The software architecture of the testbed provides the capability to implement real time CPS control algorithms to interact with and operate an autonomous vehicle within a connected simulator.

Autonomous Vehicle Simulator: The autonomous vehicle simulator utilized in our testbed is the TORCS Racing Simulator [163]. TORCS can be run on Windows, Linux, and Mac computers, but for our setup we have the simulator running on Ubuntu 16.04. A socket based communication is provided to access variables in the simulation, but we built a customized python API interface for easing variable access from external processes in our testbed. The simulator can be customized to output sensor values such as lidar, speed, brake, gear, track position, distance from start position, vehicle heading, and position in the race. Among the outputs, the user can change variables such as steering, acceleration, braking, and gear value.

CPS Controller: The software for both the neural network and safe controller exists on the NVIDIA Jetson TX2 board. This board is configured with the Linux4Tegra 28.2 operating system, GPU libraries such as CUDA, and machine learning libraries such has Tensorflow. The operating system is additionally patched with the RT-PREEMPT patch. This patch allows for specifying real-time priorities of executing processes at the application layer. Then priorities are then a kernel level rate monotonic scheduler which handles the sharing of resources. The configuration manager has the highest priority, while the executing processes within the DBT are premptible with a lower priority. Furthermore, buffer overflow vulnerabilities are inserted to test the effect of a code injection, and code reuse attack on the overall system behavior.

Communication: To support automotive applications, multiple communication interfaces are included such as Ethernet and CAN bus. For Ethernet communication, the ZeroMQ (ZMQ) communication library in utilized. Additionally, for the CAN bus communication, an open source library called SOCKETCAN is utilized to support the communication between the control code and ECU cluster.

III.5.2 Case Study

To demonstrate the capabilities of our security architecture, an autonomous vehicle case study is utilized. The case study is based on a platoon scenario, with one manual vehicle driving as the leader and an autonomous vehicle as the the follower. For the purpose of evaluation, the follower vehicle will be the center of focus. The follower vehicle system is composed of an ECU cluster containing sensors such as lidar, heading, and speed sensors, as well as actuators like steering, and throttle. A neural network is utilized as a vehicle controller to take lidar, brake, gear, and speed data as input while outputting actuation to control the steering, and acceleration of the vehicle. Additionally, in the event of a cyber-attack, a safe PID controller is utilized. This controller will be less optimal from a physical control standpoint compared to the neural network, but will be designed in a manner to ensure a higher degree of security, and safety. The goal of the case study is to keep the car in a safe state (center of the road), while maintaining a stable speed and distance from the leader vehicle. To assess the effect of our security architecture, several metrics are analyzed including controller execution times, recovery time (system downtime/availability), vehicle position (distance from center of road), and the vehicle damage.

Neural Network Controller: The neural network controller is built as a sequential model. The neural network architecture consists of 5 layers with 20 nodes each. The model takes a vector of 9 lidar sensor values, speed, brake value, and gear value and produces a vehicle control sequence as output consisting of a throttle and steering value for the car. This model is trained utilizing 10 hours of manual car driving data from the autonomous car simulator. The model produces consistent behavior of the car safely driving around the track at approximately 80 mph following the leader car which serves as a good baseline of operation for our security architecture. It is important to note that the controller lidar input processing function includes non-bounded input presenting a buffer overflow vulnerability on the controller.

Safe Controller The safe controller is a simple PID controller that computes the vehicle steering, and acceleration based on the speed of the vehicle, as well as as the Lidar data, and vehicle heading. The controller aims to keep the vehicle in the center of the road. The assumption is made that this controller has been proven to be fully secure, and the adversary can not perform exploitation.

Additional Vehicle Processes In addition to the vehicle driving controller, multiple external controllers are implemented. These controllers include a remote function actuator, and telematics control unit. As such, they present an additional overhead to the system that must be taken into account when scheduling. Furthermore, these controllers provide for an external communication interface that opens up an avenue for remote exploitation. The telematics control unit is responsible for relaying vehicle information such as speed, distance, and damage to a remote database representing a central operating station for emergency and maintenance personnel. The remote function actuator is responsible for determining the presence of a vehicle key fob, by sending a constant message signal to the driving controller for polling purposes. With the absence of this signal, the vehicle has built in logic to shut off and terminate operation.

CAN Bus Message Synchronization Since a CAN BUS utilizes a broadcast based communication method, transmitted messages must be syncrhonized to ensure that packets are reliably received. [164] notes that the worst case message transmission time for an 8 byte CAN packet was found to be 138 microseconds. As such, a CAN timeslot of 200 microseconds was chosen for synchronizing transmitted messages. For the case study, 5 different messages are transmitted in this order: the sensor input, key fob detection message, telematics sensor data message, actuation output, and telematics actuator data message. These message timeslots will combine to form a communication period of 1 millisecond. Furthermore, the 2 message gap between the received sensor input, and transmitted actuator output provides a 400 microsecond buffer for control computation. The message transmission order can be observed in Figure III.5.

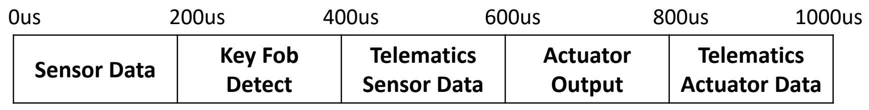


Figure III.5: CAN Bus Message Timeslots

Configuration Manager Setup: The Configuration Manager is responsible for initializing the underlying security architecture, as well as providing attack detection and reconfiguration mechanisms. For this case study, the Configuration Manager is configured to spawn two underlying child processes consisting of a neural network controller and safe controller. One instance of each will be spawned inside of a DBT enclosure which provides a customized vitualized environment including ISR with AES 256 encryption, and fine grained ASR at function level granularity. The neural network controller will be assigned to execute by default, while the safe controller will assigned the role of backup controller, remaining in a waiting state. The detection algorithm is configured to be triggered by an invalid instruction or invalid address exception caused by an attack failure due to the MTD defense mechanisms. Upon attack detection, the reconfiguration algorithm will transfer execution to the backup safe controller and spawn a new neural network controller instance with a new randomization environment. Upon the vehicle reaching a stable state, execution will then be transferred back to the neural network controller.

III.5.3 Attack Scenarios

For this case study we focus on exploits that rely on buffer overflow based vulnerabilities. Two of the most common exploits in this class are code injection and code reuse attacks. During these two scenarios, the adversary will leverage unsecure communications between the remote function actuator and the neural network driving controller to inject a malicious payload into a vulnerable input buffer. This buffer was manually inserted into the high performance CPS controller to aid in the evaluation process. At this point, the attacker can either execute customized code on the stack, or can redirect control flow to other existing points in the program to disrupt safety-critical behavior. The below scenarios will be run under three circumstances for comparison 1) Baseline - Normal operation where no attacks or defenses are in place 2) Attack - An adversary executes an attack without any defenses in place 3) Defense - An adversary executes an attack, but our security architecture is in place.

III.5.3.1 Scenario1: Code Injection Attack

This scenario involves an autonomous vehicle starting out driving on a straight road. At the point where the vehicle starts to take a turn at 70 seconds into the simulation, an adversary spoofs a malicious RFA packet to exploit a buffer overflow vulnerability in the operating neural network controller, and execute a code injection attack. The spoofed packet will contain an executable instruction payload to start a malicious controller that transmits false steering and throttle messages to cause the vehicle to drive straight at full speed, failing to turn on the curve, and consequently driving into a wall.

III.5.3.2 Scenario2: Code Reuse Attack

This scenario starts off the same as the first scenario with an autonomous vehicle driving on a straight road and then turning on a curve. The adversary leverages a buffer overflow vulnerability in the neural network controller found through reconnaissance efforts and spoofs a malicious packet as input to the buffer at 70 seconds into the simulation. Instead of executing code directly on the stack like in scenario 1, the attacker will craft the exploit specifically to overwrite the return address of the current controller function to redirect control flow to an existing safety-critical function in the program that causes the vehicle to turn left. By continuously redirecting control flow back to this function, the vehicle will move into a state of continuously turning left in circles. The goal of the attacker is to put the vehicle in this state with the hope of causing a crash into a wall, or by approaching vehicles from behind.

III.5.4 Overhead Results

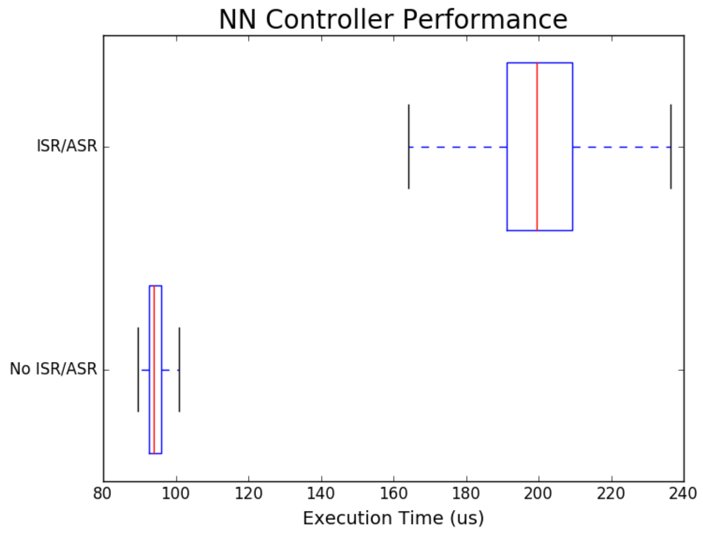


Figure III.6: Neural Network Controller Execution Times

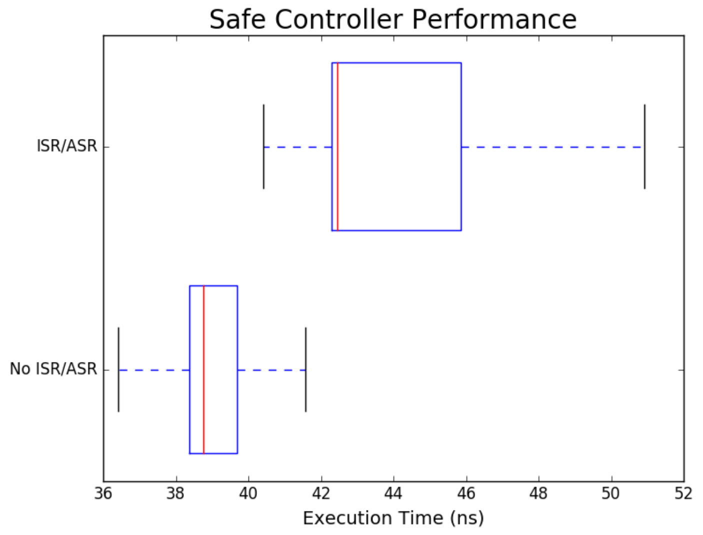


Figure III.7: Safe Controller Execution Times

As the target sampling rate is 20 Hz, requiring a 50 ms deadline, it is critical to have a low overhead in respect to the security architecture. To accurately measure the overhead of our architecture, we measure the time taken between the CPS controller receiving sensor input, and transmitting actuation output. This time difference represents the amount of time taken for computation by the controller. We repeat this process for 1000 iterations of the controller with varying inputs to identify an average execution time for the controller process. By measuring the average execution times for the CPS controller without our architecture, and with our architecture, we can have a relative comparison of the overhead that our architecture presents.

When observing Figures III.6, and III.7, the overhead created with both ISR and ASR enabled is minimal enough to maintain execution times under the respective real time deadlines. For example, when looking at the low complexity controller (safe controller) execution times, overhead is about 10.2%, bringing the average execution time from approximately 39 nanoseconds to 43 nanoseconds. Additionally, this overhead brings the worst case execution time from 42 nanoseconds to 51 nanoseconds. These results represent the lower bound of our architecture overhead. When looking at the complex controller (neural network controller), we can obtain a more accurate representation of the upper bound of the overhead. In this case, the average execution time will increase from approximately 100 microseconds to 210 microseconds, a 110% overhead. The worst case execution time will consequently increase from 267 microseconds to 580 microseconds. However, even with a scaling factor of 10, this is still well under the 50 millisecond deadline, leaving room for scheduling other complementary tasks in the rate monotonic scheduling algorithm.

III.5.5 Worst Case Recovery Time

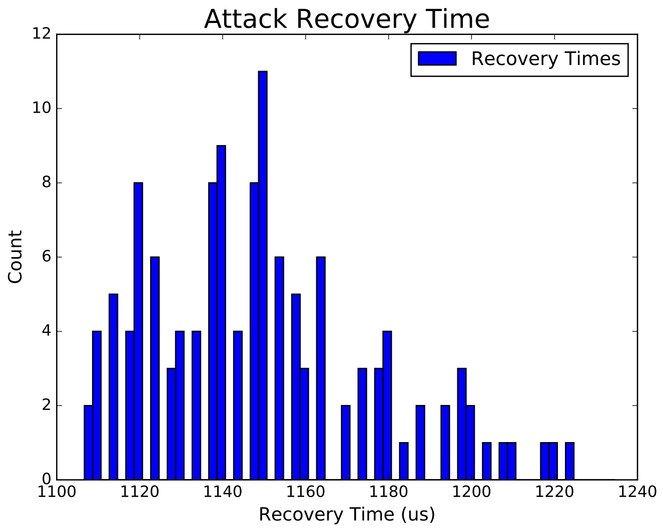


Figure III.8: Attack Recovery Times

It is not only important to meet the real time deadlines under normal circumstances, but it is equally as critical to meet deadlines when a cyber-attack occurs. As such, during an attack scenario, the attack must be detected and the architecture must reconfigure fast enough to meet the appropriate real time deadline, and consequently maintain safety and stability of the controllers. Figure III.8 illustrates the respective recovery times of the complex and safe controller. To measure the recovery time, we recorded the time difference between the last actuation transmission and when the backup controller sends the next actuation transmission after resuming execution.The average recovery time observed is approximately 1.158 ms, while the worst case observed was 1.230 ms. This means that in all of the experimental iterations, the architecture is able to recover in time to meet the respective deadline. However, when assessing the absolute worst case scenario, the cyber-attack will occur close to the end of the period. With a worst case safe controller execution time of approximately 52 ns, the next actuation command will be ready at the time $50ms + 52ns$ after the last actuation command, essentially equating to just after the next period starts. This means that in the worst case scenario, the recovery process will miss at most one deadline. During these circumstances, a fail safe mechanism is implemented in the Configuration Manager to send the last actuation command to the physical plant until the safe controller fully takes over execution.

III.5.6 Simulation Results

Figure III.9 illustrates the vehicle position relative to the center of the road with respect to the code injection attack scenario. At approximately 70 seconds into the simulation a malicious payload is injected in an attempt to hijack control of the vehicle. In the case where MTD defense mechanisms were not enabled, the payload successfully spawns a malicious controller that results in the vehicle driving off of the road and crashing into a side wall at approximately 80 seconds. At this point, the vehicle will sustain damage and skid along the wall until reaching a complete stop around 120 seconds. However, when looking at the code injection scenario where MTD defense mechanisms are enabled, once the payload is injected, successful recovery to the safe controller occurs, providing control stability while the vehicle is driving on the track curve. Once the vehicle reaches a more stable state (straight road), at 95 seconds, the neural network controller will resume execution, and the vehicle behavior becomes more closely aligned with the baseline scenario.

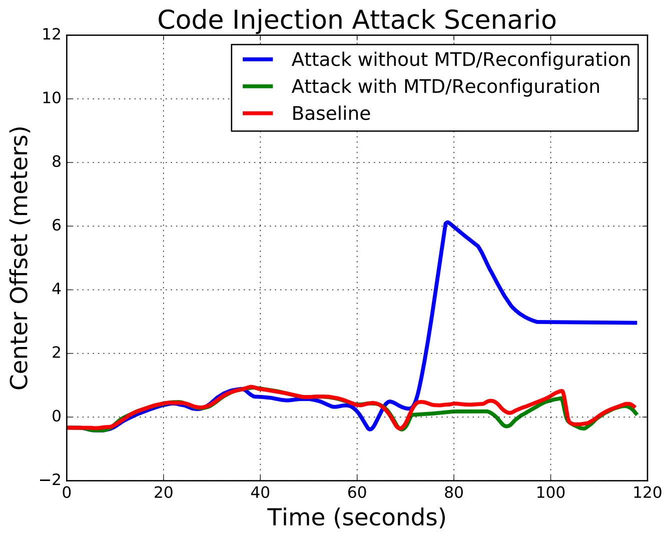


Figure III.9: Vehicle Road Center Offset Time Plot

In the second scenario where a code reuse attack is executed, Figure III.10 illustrates the vehicle distance from the center of the road. At 70 seconds into the simulation, the payload is injected into the vulnerable input buffer. At this point, when ISR and ASR are not enabled, control flow is successfully redirected to the turn left function, causing the vehicle to constantly move left in a loop, and explaining the oscillating distance behavior in the plot. However, when ISR and ASR are enabled, the attack will fail due to an invalid memory exception, and recovery will occur to the safe controller. Similarly to scenario 1, once the vehicle reaches a more stable state past the curve in the road, reconfiguration then transfers back to the newly spawned neural network controller for the rest of the simulation.

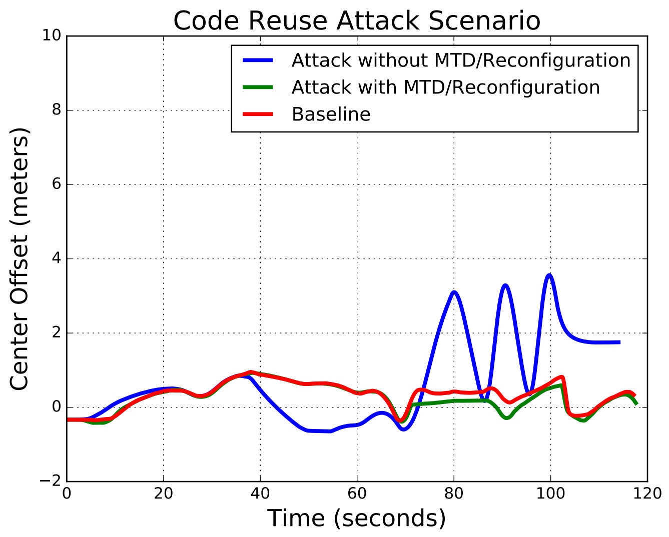


Figure III.10: Code Reuse Scenario Road Center Offset Time Plot

III.6 Evaluation 2: Udacity Image Based Self Driving

An autonomous vehicle case study is utilized to demonstrate the capabilities of the developed security architecture. The system is comprised of electronic control units controlling steering, and speed actuation, while receiving forward facing camera images as input. The autonomous vehicle controller consists of a neural network controlling the steering angles based on input camera images. The goal for the case study is to keep the car driving on the road while maintaining a safe state of operation.

Neural Network Controller: The neural network controller is built based on the NVIDIA recurrent neural network model [165]. The neural network architecture consists of 9 layers including a normalization layer, 5 convolutional layers, and 3 fully connected layers. Overall, there are over 25 million nodes in the neural network.

The neural network model takes a 66 $\times$ 200 pixel RGB camera image of the view from the front center of the car and produces a vehicle control sequence as output consisting of a throttle and steering value for the car. This model is trained utilizing 8 hours of manual car driving data from the autonomous car simulator. The model produces consistent behavior of the car safely driving around the track which serves as a good baseline of operation for our security architecture. The neural network controller includes three component threads: a timer driven publisher that transmits vehicle control messages to the simulator, an event driven subscriber that obtains new camera images from the simulator whenever data is updated, and the controller which obtains new vehicle control data by passing the camera image through the neural network model. It is important to note that the subscriber function includes non-bounded input presenting a potential buffer overflow vulnerability and the possibility of a successful code injection attack on the controller.

Controller Configuration: Each controller is implemented as a real time process operating on the Linux RT-Preempted patched kernel. As such, each controller process has three concurrent operating threads consisting of the message publishing operation, the controller operation, and the message subscribing operation. To satisfy the safety constraints of the autonomous vehicle, the steering needs to be updated at least at a 10 Hz frequency.

- Publisher Operation - Transmits the computed steering angle back to the autonomous vehicle simulator. The target frequency is 10 Hz, attempting to send a steering angle once every 100 milliseconds.

- Subscriber Operation - This operation is event triggered based on receiving messages from the autonomous vehicle simulator containing the respective GPS coordinates, direction of the car, and camera image. The subscriber function updates the process input variables every 100 seconds, targeting a 10 Hz frequency.

- Controller Operation - Takes in camera input as input and computes the steering angle as an output. Targets a 10 Hz frequency, updating the steering angle every 100 milliseconds.

Configuration Manager Setup: The Configuration Manager serves as the parent process of our MTD architecture. The main functionality is to oversee the execution of the subsidiary controller processes, tying in the ability to detect attack instances through program exception handlers. For this case study the Configuraiton Manager oversees the operation of two autonomous driving neural network controllers. Each controller has a different generated randomization key. Due to load time concerns of the controller, the backup neural network controller is started at runtime in an idle state, waiting for a recovery event. In the instance of attack, the Configuration Manager transfers control from the neural network controller to the backup controller and a new neural network controller is started in the place of the attacked controller. This controller additionally is assigned a unique randomization key, limiting the vulnerability to side channel attacks and adversarial reconnaissance efforts.

Comparison Metrics: Several different metrics are utilized to compare the effectiveness of our security architecture. These metrics focus on the areas of security, as well as physical behavior. From a security standpoint the categories of integrity and availability are looked at. The goals of our architecture are to maximize the integrity of a system by protecting against cyber-attacks, while maximizing the availability to keep the system from crashing and perform optimally. To analyze integrity and availability, the architecture effectiveness against attack attempts, as well as the recovery downtime and architecture performance overhead are measured. To analyze the performance we measure the execution times of our CPS controller both without and with our randomization environment. To analyze the recovery downtime, we make the assumption that the detection time is negligible. Therefore, we measure from the time of attack detection, to the time the backup controller takes over execution. Finally, to measure the resulting physical behavior of the system, the distance from the center of the road is utilized to determine how safe of a state the vehicle is in.

III.6.1 Experiment Setup

A hardware-in-the-loop testbed is necessary for determining, measuring, and analyzing the effects of cyber-attacks on real CPS. As a part of this platform, real embedded hardware is utilized with both sensing and actuation interactions with the physical system. However, for many systems such as autonomous vehicles, building the real CPS is not feasible due to financial, logistical, and safety reasons. As such, it is common to use physics simulators to act as the physical plant with the embedded hardware providing the computation and communication capabilities of the CPS. We use a custom hardware in the loop testbed to effectively evaluate our MTD security architecture with developed experiments. Our testbed includes hardware-in-the-loop embedded hardware combined with an open source simulation workstation for creating experiments to measure the effects of cyber-attacks and defenses on the safety, security, and reliability of autonomous vehicles. An overview of our testbed is shown below. For a more detailed description of our testbed, please go to Appendix A.

III.6.1.1 Hardware Architecture

Our autonomous vehicle testbed allows for developing distributed CPS scenarios for analyzing various metrics. The testbed includes embedded hardware serving as the CPS computation platform, a simulation workstation serving as a model of the physical environment and providing sensing and actuation capabilities, and a network providing communication capabilities both within the distributed CPS architecture, and to the simulation workstation. The testbed computational hardware includes a NVIDIA Jetson TK1 board [161]. The NVIDIA Jetson includes both a 2.32 GHz ARM quad core Cortex-A15 CPU along with a GK20a GPU with 192 SM3.2 CUDA cores. The ECU cluster is comprised of Beaglebone Black 1 GHz ARM Cortex-

A8 embedded computing boards [162] serving as the sensors and actuators of the system. The simulation workstation consists of a single i7 desktop computer with a 7200 RPM hard drive. Finally, the network infrastructure consists of two networks: a standard 100 Mbps ethernet TCP/IP network for communication from the hardware-in-the-loop testbed to the simulation workstation and a 1 Mbps CAN Bus network between the computational platform and ECU cluster. The hardware architecture is illustrated in Figure III.11.

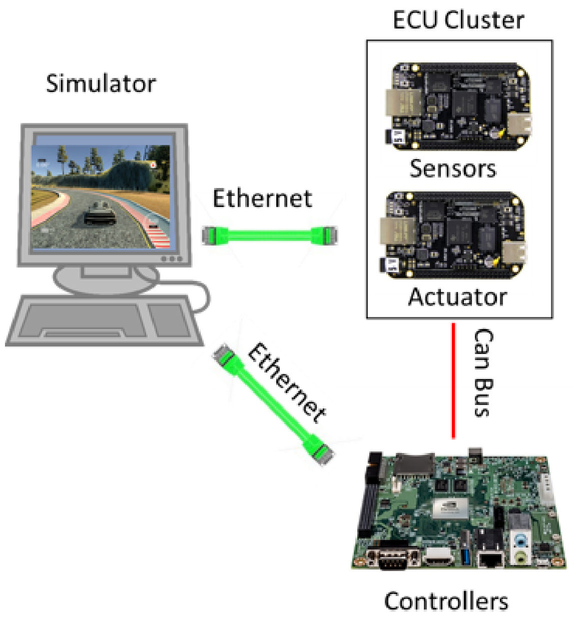


Figure III.11: Testbed Hardware Architecture

III.6.1.2 Software Architecture

The software architecture of the testbed provides the capability to implement real time CPS control algorithms to interact with and operate an autonomous car within a connected simulator.

Autonomous Vehicle Simulator: The autonomous vehicle simulator is developed based off of the Udacity autonomous car open source simulator which was built utilizing the Unity Game Engine [166]. The simulation workstation utilizes a Windows 10 operating system environment which is used to build and run the autonomous vehicle simulator executable. The simulator utilizes an API based on the SocketIO library. Each simulator variable can either be accessed or updated through JSON. By default the simulator outputs vehicle speed, and an image of the car front-center view of the road ahead. The simulator code was edited to provide additional GPS coordinates, and a unit vector describing the direction orientation of the car. The simulator takes as input a steering angle and throttle value which can be determined through an external control script.

CPS Controller: The CPS control code is developed on the NVIDIA Jetson TK1 embedded platform.

The NVIDIA Jetson is configured with the Linux4Tegra operating system, applicable GPU libraries such as OpenGL, and CUDA and machine learning libraries such as Tensorflow. To enable real time support, the Linux kernel is patched with the RT-PREEMPT patch. This update converts Linux into a fully preemptible kernel and produces response times within the microsecond range. Furthermore, the control architecture provides support for dynamic binary translation utilizing the MAMBO environment [56]. MAMBO creates a virtual layer that provides the capability to edit ARM machine code before it reaches the processor for execution. As such, instruction set randomization (ISR) support is provided by first randomizing executables and dynamicaly derandomizing instructions at runtime as they are fetched. Additionally, the controller software architecture is built modularly emphasizing the distributed component nature of CPS. Finally, vulnerabilities are built into the existing implementation to support attack model experiments. For example, vulnerabilities such as non-bounded input copy functions, and a lack of stack protections are inserted to test the effect of a code injection attack on the overall system behavior.

Communication: The ZeroMQ (ZMQ) communication library is utilized for providing distributed messaging between the autonomous car simulator interface and the controller code. This communications allows for incorporating sensing, and actuation capabilities into CPS control code. ZMQ utilizes a publisher-subscriber methodology that allows components to publish output messages to various components (ip addresses-ports) throughout the network and subscribe to receive message data from other components as well as filter communications based on a specific topic. The SOCKETCAN communication library is utilized for CAN Bus communication between the computational control code and ECU cluster.

III.6.2 Attack Scenarios

The following attack scenarios are built upon the code injection attack technique discussed in Section III.2. The adversary takes advantage of unsecure communications between the front facing camera and neural network controller, as well as a buffer overflow vulnerability within the image input processing function in the neural network controller to execute external code payloads on the stack.

III.6.2.1 Scenario1: Code Injection Attack on Straight Road

In the first scenario a code injection attack is utilized to demonstrate the capabilities of our security architecture. A straight road example is used to demonstrate the recovery process in the event of leeway in the physical behavior safe operation. In the default configuration of the autonomous vehicle controller, a neural network is used to control the steering angles based on inputted camera images. However, the neural network controller contains a buffer overflow vulnerability in the camera image processing code. As such, the attacker can leverage this vulnerability to inject a remote-shell payload to divert program execution to the attacker. At this point the attacker has full remote root access to the system, providing unlimited opportunities to damage the car. For the scenario, the attacker executes this remote shell attack at 20 seconds into the simulation. Then, a malicious controller is executed remotely to spoof control packets to the steering and speed electronic control unit of the car. For demonstration purposes the malicious controller causes the vehicle to drive straight ahead at full speed, eventually driving off of the road. When run under our security architecture, the attacker payload will fail and the system will recover to a backup neural network controller. In the following results a comparison is made between this attack scenario run with no defense mechanisms and under our security architecture.

III.6.2.2 Scenario2: Code Injection Attack on Curved Road

The second scenario is built off of the first scenario in that a code injection attack is utilized to demonstrate the operation of our security architecture under adversarial conditions. However, in this case a curved road example is used instead of the straight road used in the previous scenario. This provides a more unstable control situation where it is more crucial for the system to recover quickly to regain vehicle stability. Any failure in accomplishing this will lead to the vehicle driving off of the road faster than in the case of the straight road. For the attack demonstration, a adversary injects a malicious payload at 90 seconds into the simulation through a buffer overflow vulnerability in the camera input processing functionality. Then, the payload opens a remote root shell, allowing the attacker to run a malicious controller that drives the vehicle straight at full speed. As in the case of scenario 1, when our security architecture is implemented, the attack will fail and the system will recover to a backup neural network controller. However, it is crucial that this recovery process is as minimal as possible, due to the need to keep accurate steering control of the vehicle on the curve.

III.6.3 Results

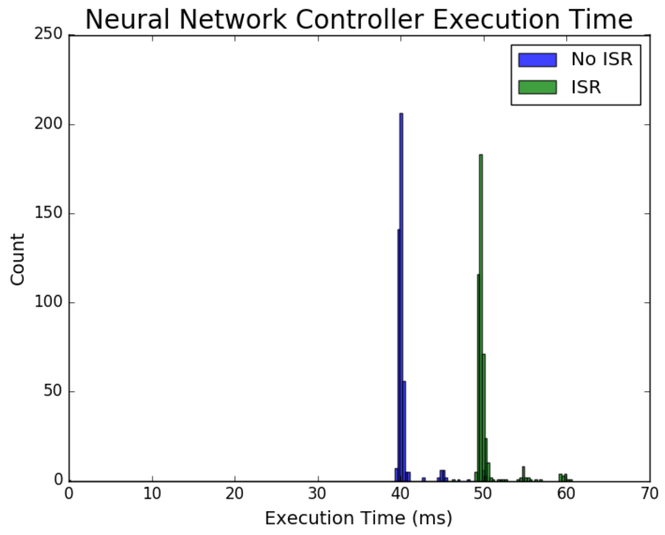


Figure III.12: Neural Network Controller Execution Times

In respect to the two attack scenarios, the target real time deadline is 100 ms correlating to a 10 Hz controller frequency. Therefore, it is the goal that the security architecture overhead is low enough that the controller execution time remains under this threshold. When observing the execution time of the neural network controller at baseline without ISR enabled in Figure III.12, the average execution time is 40.44 milliseconds which is comfortably under the deadline. When ISR is enabled under the security architecture using dynamic binary translation, the average execution becomes 50.24 milliseconds. This value and the maximum recorded execution time is still comfortably under the target deadline. Additionally, the average recovery time with and without the randomization is approximately 10 ms with no significant overhead caused by our security architecture. When factoring in the fact that the backup controller will take at most one full period (100 ms) once started to compute and transmit a steering angle message to the actuator, the system will at worst fully recover before the second deadline (200 ms), missing only one real time deadline. The distribution histogram of the recovery times is illustrated in Figure IV.11. The amount of missed deadlines during recovery is illustrated in Figure III.14.

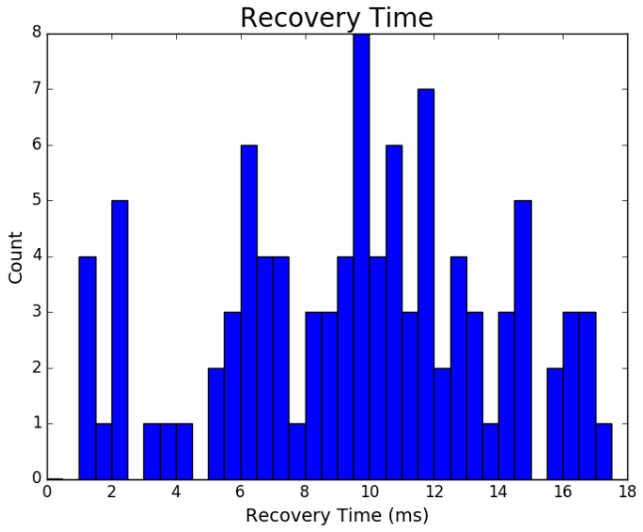


Figure III.13: Recovery Times

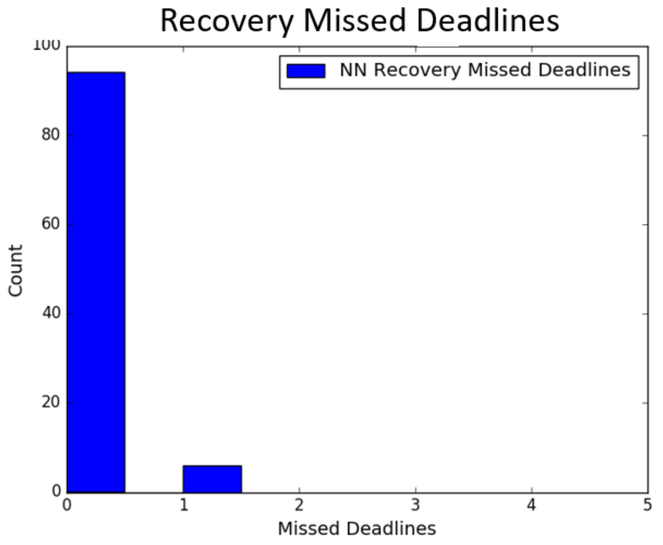


Figure III.14: Recovery Missed Deadlines

In the event that a code injection attack occurs in scenario 1 without ISR enabled, the attacker is successfully able to obtain a root terminal and execute a malicious controller to drive the car off of the road. As such, in this case the car will drive further and further away from the center once the road starts to segment away. However, in the case where the security framework is enabled with ISR, the payload fails and the car successfully recovers to a backup neural network controller keeping the path in line with the center of the road. Therefore, in this case the distance the car travels away from the center of the road will be bounded. This behavior is observed in Figure IV.12.

It is additionally observed that the center offset during the no attack success scenario is bounded to approximately 1 meter and has a significantly smaller standard of deviation compared to the attack success scenario. The average center offset of the no attack scenario is .57 meters while the average center offset of the attack success scenario is 3.50 meters. It is also important to note that in this experiment simulation, the car crashed into a rock at around 5 meters away from the respective center of the road. In the case that there are no obstacles in the path of the car, the distance off of the road will become unbounded.

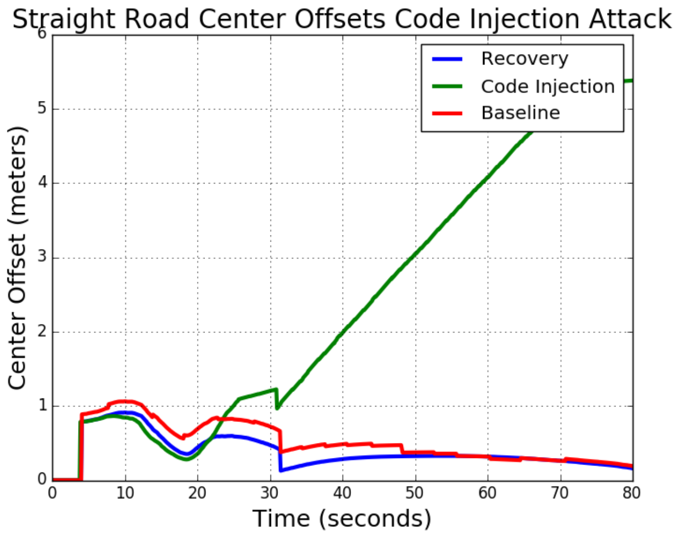


Figure III.15: Code Injection Attack Straight Road Center Offset Time Plot

In the event that a code injection attack occurs in scenario 2 without ISR enabled, the attacker is successfully able to spawn a root shell and execute the malicious controller. As such, in this case the malicious controller spoofs control packets to make the vehicle drive straight at full speed. Since the road is curved, this behavior results in the vehicle driving straight off of the road. However, in the case where the security architecture is enabled with ISR, the divertion attempt fails and the car successfully recovers to the backup neural network controller reacting quickly enoug to keep the vehicle on the road and readjust to the center of the road. This behavior is observed in Figure III.16.

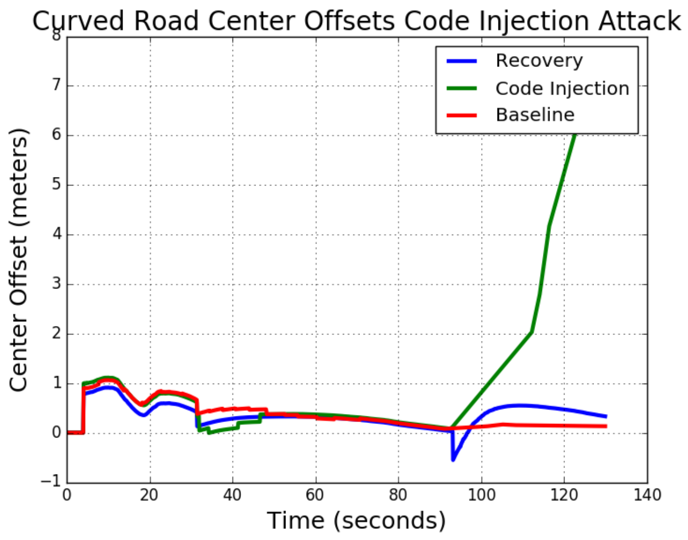


Figure III.16: Code Injection Attack Curved Road Center Offset Time Plot

It is observed that the center offset during the no attack success scenario is bounded to approximately 1 meter and has a significantly smaller standard of deviation compared to the attack success scenario. The average center offset of the no attack scenario is .57 meters while the average center offset of the attack success scenario is 6.62 meters. In this experiment simulation, the car crashed into a lake at around 8 meters away from the respective center of the road. In the case that there are no obstacles in the path of the car, the distance off of the road will become unbounded.

III.7 Conclusion

In this chapter, we have successfully leveraged ISR, and ASR to protect against code injection, and code reuse attacks. We have extended our MTD security architecture to upgrade security by implementing AES 256 encryption for ISR, and further adding fine grained ASR support at function level granularity. These techniques which greatly improve security have been shown to have high but acceptable performance overhead in the autonomous vehicle case study utilized in this paper. Furthermore, attack detection, and recovery methodologies have been successfully integrated to maintain safe system availability in the case of cyber-attacks, addressing the drawbacks of traditional MTD approaches in leading to system crashing. We describe our security architecture in terms of the high level organization, as well as the process flow of the implementation. For evaluating, our security architecture we introduce a developed hardware in the loop testbed that emulates CPS control software on hardware consistent with a distributed CPS deployment environment, with the additional ability of assessing the networked communication between the physical plant (TORCS simulator), ECU cluster, and controllers. By utilizing this testbed, we were able to obtain live measurements and analysis of the system in both normal operation, and under cyber-attacks. For the case study, we evaluated several metrics of our security architecture including controller performance overhad, system recovery time, and physical safety metrics. It has been shown that the performance overhead, and recovery time is minimal enough to support safe, and stable vehicle driving controller operation. In the following chapters, we plan on implementing data space randomization, and integrating mixed time and event triggered functionality.

CHAPTER IV

Data Space Randomization for Securing Cyber-Physical Systems

IV.1 Introduction

The design of safety-critical infrastructure is widely changing with the introduction of CPS. Traditionally isolated and standalone systems are becoming connected, utilizing communication channels to form distributed systems. These changes are beneficial for increasing the precision, consistency, and reliability of computations by allowing for more sophisticated control algorithms to be utilized. However, with the newly connected state of CPS the attack surface is also expanded. Systems were not originally designed with remote cyber-attacks in mind, creating a vast array of problems that could arise from adversary exploitation. Instead of necessitating physical access adversaries can now gain access and exploit software remotely to inflict physical consequences. In the case of autonomous vehicles, controller compromises can lead to vehicle crashes, passenger data exfiltration, and destination changes.

Vulnerabilities often exist in legacy CPS software and it is difficult to integrate state of the art security features for hardening purposes. As such, several attack vectors exist that are less common in traditional information technology systems. One commonly utilized exploit is a non-control data attack. Instead of code injection attacks and code reuse attacks which focus on redirecting control flow, non-control data attacks focus on utilizing vulnerabilities like buffer overflows to alter adjacent variables. It is popular to use this technique for bypassing password authentication mechanisms, but in CPS this can extend to altering safety-critical variables leading to potentially fatal consequences. Data Space Randomization (DSR) has become a popular moving target defense (MTD) technique for protecting against non-control data attacks. By altering the representation of critical variables at runtime, any attempt to overwrite will result in an outlier data value.

In existing DSR approaches success is defined by the translation of adversary injected values into outlier data. However, if this data is still of a valid representation it will not result in any program exceptions and even could possibly satisfy existing detection constraints if the translated value happens to fall within a defined safe range. Additionally, existing approaches rely on source code transformations, but legacy software usually can only be accessed in a binary format. Due to the safety-critical nature of CPS and sheer volume of legacy code, these existing solutions fall short of being effective, allowing for unsafe repercussions to occur from cyber-attack attempts. As such, it is important to develop a methodology for performing DSR at the binary level, allowing for a dynamic randomization process at runtime, and providing re-randomization capabilities to further hinder adversary reconnaissance efforts and maintain availability.

In CPS, timing is a critical component to maintaining safe behavior. A majority of software consists of real time constraints that are relied upon for the integrity of scheduling processes. Regardless of hard or soft real time constraints, a failure in meeting deadlines can result in dangerous consequences, especially in safety-critical CPS such as autonomous vehicles, aircraft, and medical devices. Therefore, when considering adding defense mechanisms such as DSR, it is not only important to be effective against cyber-attacks, but it is equally as important to minimize the overhead as much as possible, limiting the likelihood of missed deadlines.

The main problem that arises in this chapter is how do we protect against non-control data attacks at the binary level while determining variable integrity violations. Our hypothesis is that by utilizing DSR in combination with static analysis and variable redundancy checking, we can protect against non-control data attacks, while detecting instances of variable integrity violations. Furthermore, by using variable key storage within program memory we hypothesize that we can limit the overhead of DSR, maintaining real time constraints in safety-critical CPS. Finally, by detecting variable integrity violations, we can implement reconfiguration to transition execution to a new backup controller to ensure that safe operation, and availability is maintained within the CPS.

ISR, and ASR protect against code injection and code reuse attacks. However, because these MTD techniques fail to protect against non-control data attacks, DSR becomes a critical defense. We create a software DSR implementation utilizing dynamic binary translation at runtime to randomize critical variables, and derandomize them for memory accesses. Additionally, detection capabilities are integrated to leverage a variable redundancy structure to identify instances of attacks. Finally, we integrate our approach with reconfiguration to transition execution to a backup safety controller during an attack attempt. The contributions of this chapter are as follows:

- We develop a DSR runtime approach using dynamic binary translation for the purpose of randomizing, derandomizing, and detecting cyber-attacks at runtime
- We develop an attack detection approach to utilize a comparison of redundant variables with different randomization keys to identify a compromise of integrity.
- We develop a reconfiguration scheme to ensure that safety and availability is maintained during a cyber-attack attempt.
- We implement our security architecture on a developed hardware in the loop testbed using a combination of off-the-shelf embedded computing hardware and open source simulation software
- We present an autonomous vehicle case study to demonstrate the effectiveness of our security archi-

tecture in limiting the impact of cyber-attacks in the context of an advanced emergency braking system (AEBS) scenario.

IV.2 Problem Formulation

With the introduction of CPS such as connected and autonomous vehicles, traditionally standalone systems are now becoming significantly reliant on software infrastructure and remote communication interfaces. Current automobiles include over 100 million lines of code and 50 to 70 electronic control units (ECUs), similar to the level of a F35 fighter jet [22]. Due to the large investment required for redesigning a system from the ground up, automotive companies often attempt to build security on top of existing infrastructure, leaving a large amount of legacy code in the process. As such, attackers can leverage the large attack surface and lack of CAN bus authentication to gain entry to automotive networks, pivot to safety-critical ECU's, and disrupt the physical actuation of the vehicle.

One of the most significant vulnerabilities discovered from legacy code is the buffer overflow. Buffer overflow's result from the absence of a limitation of the length of stored input in the C and C++ language, leading to the overwriting of adjacent memory locations on the stack. By overwriting adjacent memory locations, adversaries can inject instruction payloads directly (code injection [153]), redirect control flow to existing functions in the program (code reuse [154]), and overwrite adjacent program variables (non-control data attacks). Unlike code injection and code reuse attacks, it is more difficult to protect against and detect a non-control data attack due to the minimal change in program execution.

Through vulnerabilities like buffer overflows, attackers can manipulate non control program variable data to alter program behavior without altering control flow. One common technique utilized to disrupt these types of attacks is DSR. DSR changes the internal or external representation of an application's data in such a way as to ensure that the semantic content is unmodified but unauthorized use, access, or modification is hindered [2]. For this to be accomplished the format, syntax, encoding, and other properties of the data can be randomized.

As such, DSR acts similarly to ISR in using a key based randomization and de-randomization process to encode variable data sensitive to attack. Each variable data object is randomized before it is written to memory and is derandomized after it is read from memory. Consistent with the ISR process, the randomization process can be accomplished by using an XOR operation with a randomization key [64; 65]. Additionally, there is also the possibility of using other symmetric encryption algorithms such as those in the AES family to add further security to the application. DSR provides both the ability to use a common shared randomization key, but for enhanced security, each variable should be mapped to a unique randomization key.

IV.2.1 Threat Model

An exemplary vehicle system model includes 6 components: a sensor cluster, actuator cluster, driving controller, telematics control unit (TCU), remote function actuator (RFA), and RFID sensor. The sensor cluster provides critical data representing the current state of the vehicle such as the speed, and distance to upcoming objects (lidar). The actuator cluster provides the ability to manipulate the vehicular acceleration through the throttle and brake. The driving controller is responsible for performing computation based on the provided sensor cluster input, and outputting commands to the actuation cluster. In this paper, the driving controller is an AEBS controller that is responsible for braking the vehicle to avoid colliding with upcoming objects. Both the TCU, and RFA are responsible for providing the external interface for the vehicle. The TCU monitors the various metrics of the system, transmitting data to a remote operating station for maintenance and emergency purposes. The RFA is responsible for determining the presence of a key fob for allowing the vehicle to be turned on.

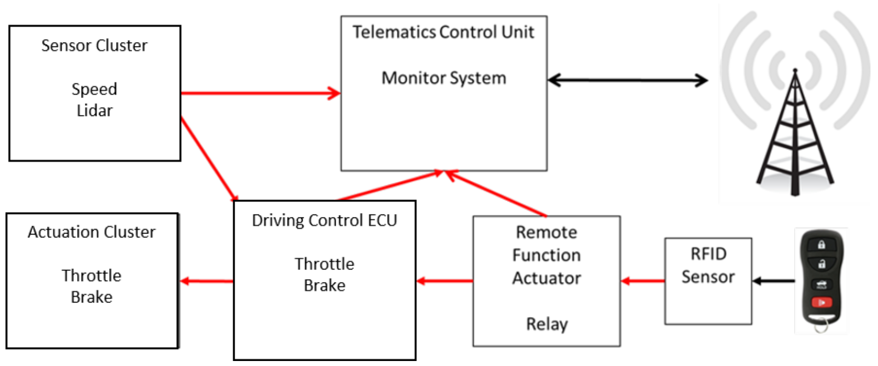


Figure IV.1: Case Study

In the system model, the sensor cluster, actuator cluster, and driving controller are on a safety-critical CAN bus network, including both communication authentication to prevent spoofing, and integrity checking within the driving controller to ensure that utilized sensor data is accurate. On the other hand, the TCU, and RFA communicate with the driving controller through a low priority CAN bus interface. Since these components are the most vulnerable to remote attacks due to being connected to external communication channels, the safety-critical and low priority communication buses protect against the TCU and RFA directly controlling the sensor or actuator ECU clusters. However, to detect the presence of the key fob, the driving controller constantly polls for status updates from the RFA. This communication is authenticated to prevent message spoofing, but there is a memory corruption (buffer overflow) vulnerability in the driving controller that provides an opportunity for memory corruption attacks.

The attack model for this paper focuses on a non-control data attack on a vehicle network. The authors in [159] note that the biggest current threat to self driving vehicles is exploitation through remote avenues.

As such, the attack vector utilized in this paper consists of the adversary compromising the TCU through the remote cellular interface, and consequently pivoting to hijack the RFA. With access to a direct communication channel with the driving controller, the adversary can craft a message payload to take advantage of the memory corruption vulnerability and alter control. At this point, an attacker can perform a non-control data attack where they can leverage the buffer overflow to overwrite adjacent safety-critical variables in the driving controller. As a result, at the next control iteration, the computation logic will utilize the attacker variable versions which could potentially effect the output and actuation of the driving controller. The goal of the attacker is to utilize the non-control data attack to cause the vehicle to perform unsafe behavior, while the goal of the defender is to prevent the non-control attack from succeeding and maintain integrity of the driving controller software.

Four assumptions are made for our approach to be successful. First, it is assumed that the sensor and actuator clusters are fully secure. The driving controller ECU may contain a the buffer overflow vulnerability utilized for control hijacking, while the TCU and RFA may contain vulnerabilities allowing for key fob message spoofing. Second, the attacker has knowledge of the relative address of a safety-critical variable relative to the start of the input buffer. Third, the attacker has knowledge of the underlying software architecture of the safety-critical controllers, allowing them to target the most impactful variables. Finally, the software can not utilize dynamic memory allocation. These assumptions are not impractical given examples demonstrated in the literature [31] as well as popular CPS standards [167; 168].

To evaluate the effectiveness of our architecture within the context of an autonomous vehicle case study, we utilize a developed hardware-in-the-loop testbed. We further utilize physical metrics such as vehicle position combined with software metrics like performance overhead in both normal operation and attack scenarios. Finally, to conclude that our hypothesis is true two observations need to be clear from the results: 1) The performance overhead needs to be minimal enough to ensure that execution times do not exceed designed real time constraints and 2) Vehicles need to follow safe driving behavior, stopping completely before colliding with the parked vehicle on the road. In the event that both of these observations are true, we can conclude that our architecture is successful.

IV.3 Static Analysis for DSR

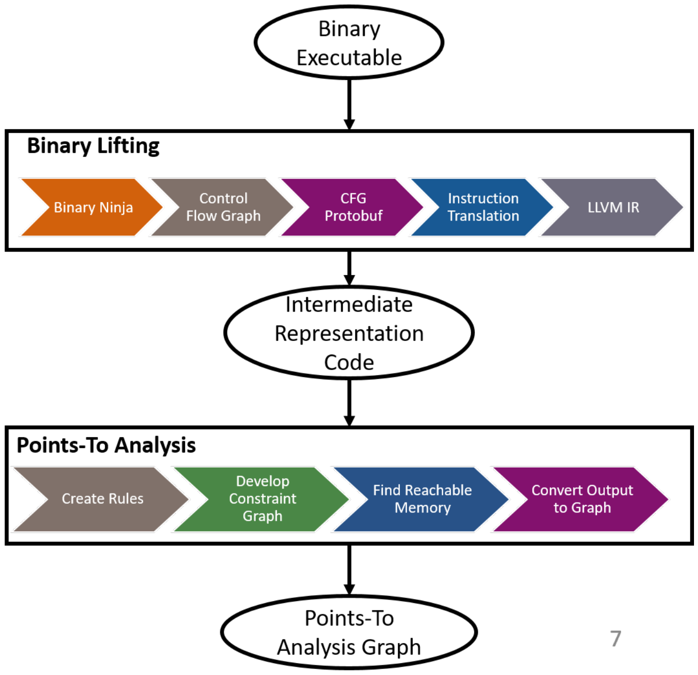
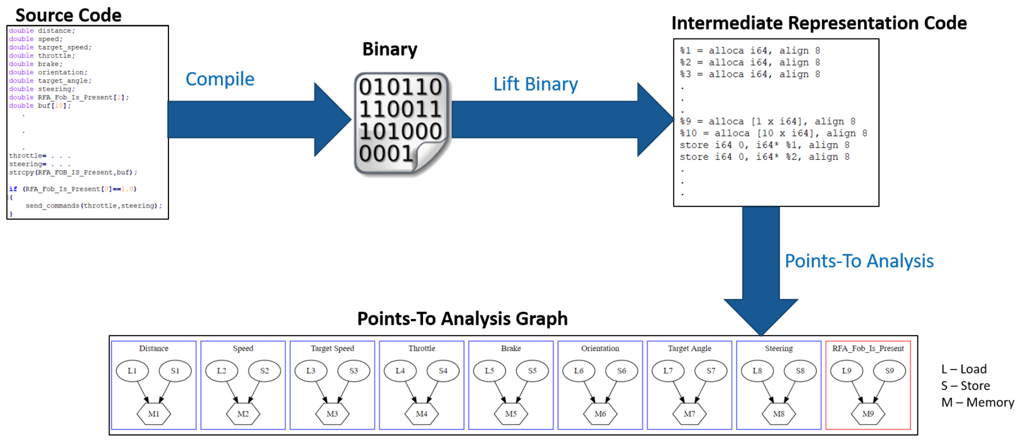


Figure IV.2: DSR Static Analysis Process

Our DSR approach is designed to operate on a native binary, eliminating the need for source to source transformations while providing the capability for automated runtime randomization and derandomization. Static variables, local variables, and heap variables can all be randomized for the purpose of protection. In cases of large programs, local variables adjacent to input buffers are prioritized in an effort to address non-control data attacks. Furthermore, by identifying the randomization variables of interest, we independently assign unique masking keys to prevent adjacent variables from being of the same encoding. In the rest of the section we describe the main segments of our DSR static analysis approach: binary lifting, points to analysis, and graph integrity checking.



```
double distance;
double speed;
double target_speed;
double throttle;
double brake;
double orientation;
double target_angle;
double steering;
double RFA_Fob_Is_Present[1];
double buf[10];
   .

   .

   .
throttle= . . .
steering= . . .
strcpy(RFA_FOB_IS_Present,buf);

if (RFA_Fob_Is_Present[0]==1.0)
{
    send_commands(throttle,steering);
}
```

```
%1 = alloca i64, align 8
%2 = alloca i64, align 8
%3 = alloca i64, align 8
.
.
.
%9 = alloca [1 x i64], align 8
%10 = alloca [10 x i64], align 8
store i64 0, i64* %1, align 8
store i64 0, i64* %2, align 8
.
.
.
```

Figure IV.3: DSR Static Analysis Example

Our DSR approach is illustrated using an exemplary vehicle driving controller component (Figure IV.3). This vehicle controller has several safety-critical variables that require randomization such as distance, speed, target speed, throttle, brake, orientation, target angle, steering, and key fob presence. Any alteration in any of these variables can result in direct safety violations in the autonomous vehicle driving behavior. As such, in the event of optimizing performance, static analysis should prioritize the variables that reflect the most damage to the overall system and surround environment in the event of their compromise.

The process of our static analysis is as followed: we start with source code for our driving controller, we then compile the source code into an executable binary, then we lift the binary code to an intermediate representation, we perform points-to analysis to create a points-to analysis graph of the binary, and finally we extract a bipartite graphs from the points-to-analysis graph representing the relationships between store and load instructions, and memory locations.

It is important to note that for our approach to work, instructions pointing to a vulnerable buffer memory location cannot also point to adjacent critical variable locations. Additionally, instructions can not point to other instructions, and memory objects can not point to instructions or other memory objects. The only valid case is an instruction pointing to a memory object. Through analysis of several example CPS programs, we have found this fact to be true. As such, we make this assumption throughout the rest of our paper.

IV.3.1 Binary Lifting

In contrast to performing source to source transformations in C programs it is difficult to manipulate and analyze variable instances in binary code. For the purpose of general analysis, it is optimal to convert the binary program into an intermediate representation (IR) format. The low level virtual machine (LLVM) compiler includes an IR representation called LLVM bitcode which we leverage for our analysis [169].

To lift a native binary to LLVM bitcode we utilize an open source tool colled Mcsema [170; 171]. Mcsema combines control flow recovery [172; 173] and an instruction translation algorithm to directly convert instructions into LLVM bitcode.

IV.3.2 Points To Analysis

The nodes extracted from the points to analysis graph can either be a load or store instruction, or a memory region. As such, it is important when designing the data randomization process to understand the relationship between these different nodes. Due to being computationally undecideable, pointer analysis algorithms generally are approximations that provide varying degrees of precision and efficiency [174]. For our approach, we rely on a context insensitive inter-procedural points to analysis implementation [175] utilizing an open source static analysis tool called SVF [176].

After points to analysis is performed, an output is generated as a points to analysis graph (PAG). After which, relationships between load and store instructions and memory regions are extracted. With this information fed in as input, unique 64 bit randomization keys are generated at load time for each respective variable object. With 64 bits of entropy, 2^{64} possible randomization keys can be generated which provides for a sufficient number of combinations for programs with a large amount of variables.

IV.3.3 Pointer Scenarios

When analyzing a legacy software binary with points to analysis, three possible scenarios can occur. The first scenario is that every encountered load or store instruction can only point to one memory location. The second scenario is that it is possible for a load or store instruction to point to multiple memory locations, but any instruction pointing to a vulnerable buffer will not point to any adjacent memory location. The final scenario is an extension of scenario 2 in that instructions can point to multiple memory locations but instructions pointing to a vulnerable buffer will also point to adjacent memory locations.

In order for our DSR approach to function correctly, scenarios 1 and 2 can occur, but scenario 3 cannot. As such, for every load or store instruction pointing to a buffer memory location where a buffer overflow can occur, those respective instructions cannot also point to another adjacent memory location. However, in the event that every instruction will point to a single memory location, or when an instruction does not point to a vulnerable buffer location but points to multiple other memory locations, our DSR approach will work. For all C/C++ controllers tested throughout our work, we have verified that scenario 1 has been the only case to occur. However, it is possible for the other two scenarios to occur in certain circumstances.

Static analysis traditionally overapproximates a set of possible memory locations, meaning that if there is a non-bounded buffer, the analysis results will reflect that the instructions accessing that buffer have the potential to access all adjacent regions. However, due to bounds analysis being included in the SVF analysis tool, the original analysis results will be pruned to reflect expected behavior, meaning that any instruction accessing the buffer will only be represented as accessing the respective buffer memory location [177].

For DSR, the main goal is to prevent a buffer overflow from effecting the integrity of an adjacent variable. The fundamental technique utilized to accomplish this goal is to encrypt every adjacent variable with a different key so that any attempt to overwrite will result in an outcome wildly different than expected. However, even though overapproximation is pruned in our PAG results, certain software designs can result in the buffer access instructions also being stated to access adjacent variables. This case can occur when a pointer in a C/C++ program is utilized to access multiple data structures including the buffer. All three scenarios are illustrated with an example in which there are two integer variables M and N, a pointer p and a buffer B which is susceptible to a buffer overflow attack.

In the first scenario, all instructions referencing variables M, N, and B will point to three disjoint memory locations. As such, when assigning randomization keys for our DSR approach, three different keys will be assigned. Therefore, in the case of a buffer overflow attack, variables M, and N will not be of the same representation as buffer B, meaning that any overwriting of adjacent variables will result in a wildly different outcome than expected by the attacker. Thus, our DSR approach will work in this case. This scenario is illustrated in Figure IV.4

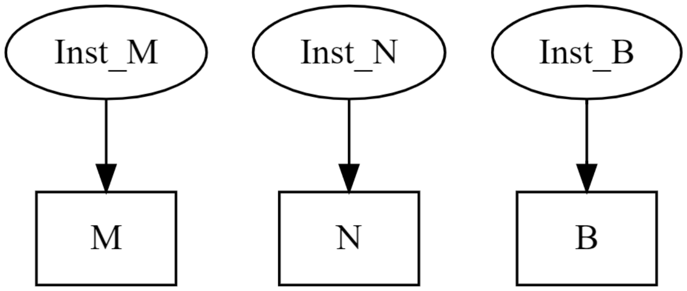


Figure IV.4: Scenario 1

In the second scenario, there will be a pointer P that points to variables M, and N, resulting in a store instruction in the PAG referencing both variable memory locations. However, this pointer will not access buffer B meaning that the store instruction in the PAG will not reference the buffer B memory location. When assigning randomization keys for our DSR approach, instead of assigning three unique values such as in scenario 1, only two unique values will be assigned: one key for the variable M and N memory locations, and a different value for the buffer B memory location. As such, in the case of a buffer overflow attack, variables M, and N, even though they have the same randomization key, will still not be of the same representation as buffer B, meaning that any overwriting of adjacent variables will still result in a wildly different outcome then expected by the attacker. Thus, our DSR approach will still work in this case. This scenario is illustrated in Figure IV.5

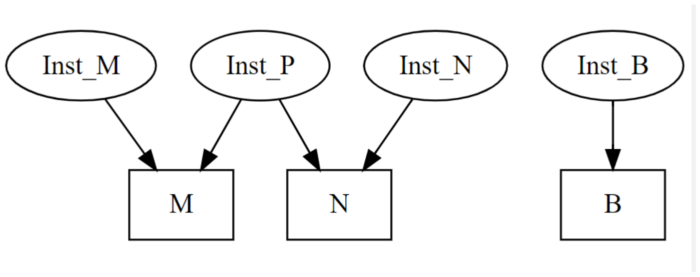


Figure IV.5: Scenario 2

In the third scenario, there will be a pointer P that points to variables M, N as well as buffer B, resulting in a store instruction in the PAG referencing all three memory locations. When assigning randomization keys for our DSR approach, only one value will be assigned, resulting in all three memory locations utilizing the same key. In the case of a buffer overflow attack, since all three memory locations utilize the same randomization key, any overwriting of M, and N from B will act like no randomization exists since the new value will be derandomized with the same key as was randomized with from the buffer. Therefore, in this case our DSR approach will not be able to protect variables M and N successfully. This scenario is illustrated in Figure IV.6

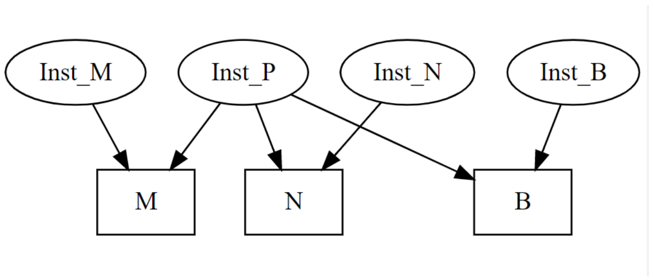


Figure IV.6: Scenario 3

IV.4 Runtime Randomization

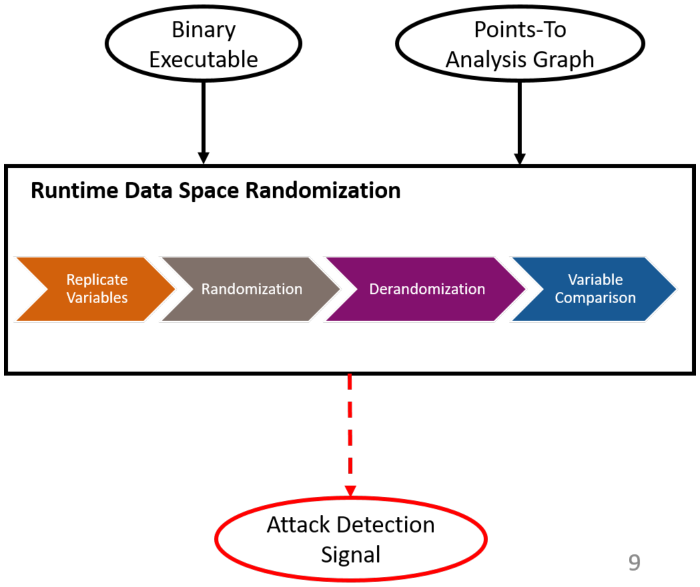


Figure IV.7: Runtime Process

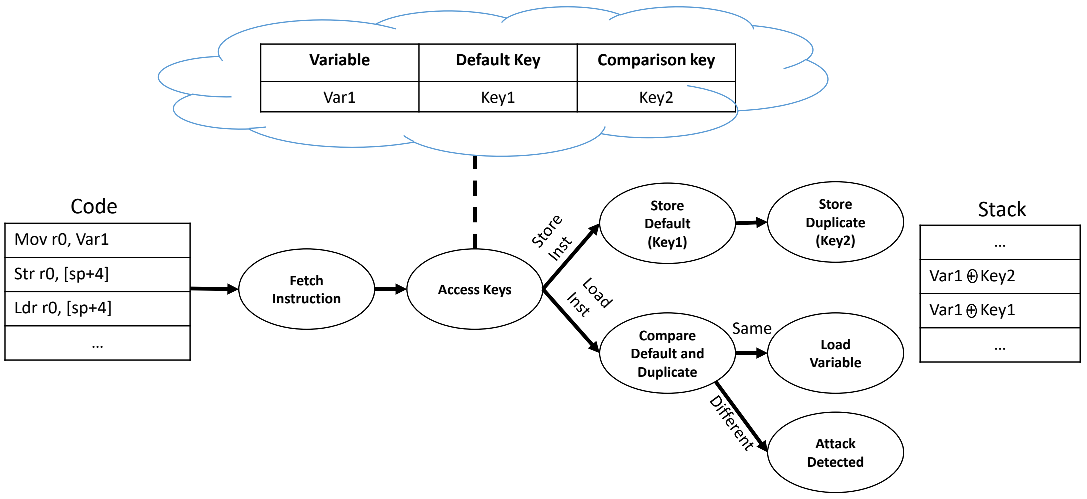


Figure IV.8: Variable Randomization Process

The main vulnerability addressed by DSR is the overwriting and manipulation of adjacent variable data to input buffers [178]. The unique randomization and derandomization of individual variables ensures that if the attacker overwrites data, the semantic effect in the program will not be of the intended nature. For example, in the case of an adversary overwriting an adjacent target speed variable for an automobile, the desired goal could be increasing the value from 65 mph to 70 mph. In this case, the adversary will leverage a buffer overflow to insert the value 70 into the target speed variable memory location on the stack. However, in the case of DSR, the input buffer and target speed variable will have different randomization keys. Due to this fact, when the target speed is read from memory, it will first be derandomized with a different randomization key than what was utilized for writing, and the resulting value read will not be 70. An important note is that the resulting data values may still be of a valid format meaning that an exception will not occur. However, the masking of variable values makes it easier for detection algorithms to determine the presence of a cyber-attack.

The primary defense mechanism utilized in our approach is the randomization and derandomization process of DSR which is illustrated in Figure IV.8. This means that when a variable is written to memory encoding will first take place with a unique randomization key, and when the variable is consequently read from memory it will be derandomized to the true value with the same randomization key before use. This means that there must be two steps inserted into the program during variable access: a randomization step during variable stores, and a derandomization step during variable loads. At load time the PAG is utilized to generate a key hash table based on the unique variables encountered during static analysis. For each variable two randomization keys will be generated, one for the default variable and one for a redundant variable instance. When a variable load statement is encountered in the program the respective variable key will first be looked up from the hash map based on the encountered address and encoded with an XOR operation. Furthermore, this table will additionally be accessed during the derandomization stage to look up the respective randomization key to perform a subsequent XOR operation on the encoded value. Since the XOR operation is a symmetric encoding technique, performing this second operation will convert the encoded variable back to the true value. It is important to note that when encoding, one XOR instruction is necessary to be inserted in the program before a store instruction, and after a load instruction an XOR instruction is inserted for decoding purposes. Additionally, encoding and decoding operations are only executed on register values, and not on the respective data in memory.

A non-control data attack consists of an adversary overwriting an adjacent variable to an existing input buffer by leveraging a buffer overflow vulnerability. During a successful attack the variable will be manipulated to a value intended to accomplish the adversaries program goal. When this variable is manipulated, DSR can cause the variable to be different than the what the adversary expects, resulting in unintended program behavior. However, in contrast to other MTD techniques such as ISR, and ASR, manipulating the variable with DSR enabled will not result in an exception due to the variable data still being of a valid format. This means that detection is not as simple as just inserting a signal handler, but a more active detection mechanism needs to be put in place.

We leverage variable redundancy for the task of detecting non-control data attacks. For example, every time a variable is written to memory a duplicate variable instance will be written in the adjacent memory location. Two different randomization keys are assigned to these respective variable instances. During a variable load operation the two variables will first be derandomized with their respective keys and compared for equivalency. In the event that they are equal, program execution can continue with variable access. However, in the event of the redundant variables being different the program is flagged for invalid behavior, execution is terminated without allowing variable access, and a backup controller resumes execution through a recovery process. By utilizing this detection technique it is not just enough for the adversary to correctly guess one variable randomization key, but must have knowledge of both keys to craft a successful payload.

When looking at the likelihood of successfully circumventing this comparison approach within a 64 bit system, the attacker would have a $1/2^{64}$ likehood of correctly representing their intended value in the default variable. However, when introducing the the second redundant variable, the attacker would now have a $1/2^{128}$ likelihood of success. Since, after attack detection a fail safe controller will take over execution, the attacker only has one chance to be successful. As such, it is extremely unlikely for an attacker to correctly randomize both variables to pass the variable comparison check. This approach ensures the integrity of our randomization process in maintained and prevents the utilization of adversary manipulated variables.

IV.5 Implementation

The key components in our architecture are the (1) CPS Controllers which control the physical plant, (2) DBT which uniquely customizes the runtime environment for each CPS controller, and (3) points to analysis graph (PAG) which describes the relationship between load and store instructions and memory regions within a program.

CPS Controller: This component is the actual software that controls the CPS application. From the most generic form, the controller takes sensor input from the system, performs computation operations, and outputs actuation commands to perform in the surrounding physical environment. Our architecture incorporates a generalist approach, allowing for a broad array of control techniques and applications to be supported. The only requirement is that the control program be in the form of native code.

Dynamic Binary Translator (DBT): This component is responsible for providing a unique randomization backend for each spawned CPS controller in the architecture. In other words, the DBT is a virtual sandbox layer that serves as an intermediary between the executing binary and the processor. The DBT has the ability to intercept instructions as they are fetched and alter program semantics before execution by the processor. As such, a DSR methodology is supported by encoding variables before storage on the stack, as well as a derandomizaton stage before loading into registers. Each variable is supported to include a dynamically generated unique randomization key, as well as a duplicate comparison variable for detecting variable tampering. Additionally, the DBT is responsible for storing a variable key mapping table which incorporates uniquely generated keys for every variable in a program. We utilize an open source dynamic binary instrumentation tool Mambo for our implementation [56].

Points to Analysis Graph: This component is responsible for identifying the relationship between variables within a program. For the implementation, the open source tool SVF is utilized which is built upon the LLVM compiler. For a PAG to be successfully generated with SVF it is necessary to translate the program binary into a LLVM IR representation. To accomplish this task we utilize Mcsema integrated with the Binary Ninja disassembler to perform binary lifting on the original program. The results of the points to analysis is fed into the DBT as input to aid in establishing the dynamic generation of variable randomization keys.

We make the assumption that the CPS controller component in our architecture is vulnerable to cyber-attacks by the adversary. The remaining components are not susceptible to cyber-attacks. As such, the variable key storage table in each DBT is assumed to be secure against integrity attacks in our threat model. Our security architecture is designed with the goal of keeping the CPS controller from becoming compromised by the attacker.

IV.5.1 Process Flow

The basis behind the execution process of our security framework is a three step approach: 1) Static Analysis, 2) Binary Load Time, and 3) Runtime. These steps are described below.

At design time, a significant amount of time needs to be dedicated to properly establishing the CPS controller. This controller is located in secondary storage and is responsible for the control of the CPS based on sensor input and actuator output. Before loading the binary for execution, static analysis is performed to analyze the relationship between program variables with Mcsema and SVF [170; 176]. At the conclusion of this step, a bipartite graphs of the relationships between load and store instructions and memory locations will be extracted from the PAG. This data structure is then iterated to identify any common instruction relationships between memory locations, and will then create a set of memory address associations. For example, if two memory locations have an instruction in common, those memory locations will be combined into a set of memory addresses which will share randomization key information.

At load time, each set of memory locations from the static analysis stage is stored in a mapping table within the DBT with two associated dynamically generated randomization keys. This lookup table will form the basis for our randomization approach during runtime. It is important to note that the generated randomization keys are unique for each DBT process. As such, there will be a different set of randomization keys for each spawned controller process.

MTD forms the backbone of our security architecture, incorporating DSR to protect against non-control data attacks as well as redundant controllers for reconfiguring during an attack. The goal is to decrease the probability of a successful cyber-attack by raising the level of effort needed by the adversary for obtaining accurate reconnaissance knowledge. Utilizing Dynamic Binary Translators, which we implement utilizing the MAMBO DBT environment [56], local program variables can be randomized at store time by XORing the value with a dynamically generated key, as well as derandomized at variable load time by executing another XOR operation with the key. Since in between variable store and load time the variable will be in an encoded state, any effort by the adversary to alter the variable will result in a derandomized value wildly different than the attackers intended change. When looking at the randomization process by the MAMBO DBT enclosure, everytime a variable store instruction is encountered two randomization keys will be retrieved from an internal variable map. These two keys will be associated with the original variable, as well as a redundant comparison variable. When storing to memory, the original variable will not only be stored in encoded form, but a redundant version of that variable will be stored in an adjacent location with a different randomization key. When a load instruction is encountered for the respective variable in the program, a check is first performed to determine if the derandomized version of both the original and comparison variable are identical. Any change by the adversary will result in a failure in this comparison, resulting in the ability to detect the attack. Once an attack is detected, the the user has many options including terminating the process or spawning a backup process.

IV.6 Experiment Setup

To enhance the ability to evaluate cyber-attack impacts in deployment environments, a hardware in the loop testbed was developed. Our testbed includes embedded hardware representing CPS software infrastructure, a simulation workstation representing the physical environment, and multiple network interfaces representing communication channels within the automobile environment. The setup of our testbed includes four components: 2 beaglebone black microcontrollers [162] representing the sensor and actuator processes in an automobile ECU cluster, a NVIDIA Jetson TX2 board [161] providing for computational power necessary for designing vehicle control algorithms, an i7 simulation desktop, and a realtime web based results dashboard. Furthermore, two communication interfaces exist including 100 Mbps ethernet, and a 1 Mbps CAN bus. The hardware architecture is illustrated in Figure IV.9.

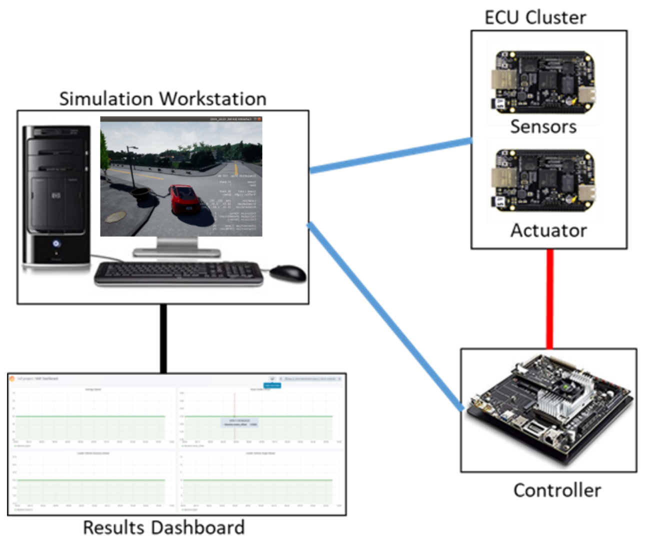


Figure IV.9: Testbed Hardware Architecture

IV.6.0.1 Software Architecture

The software architecture of the testbed provides the capability to implement real time CPS control algorithms to interact with and operate an autonomous car within a connected simulator.

Autonomous Vehicle Simulator: The autonomous vehicle simulators utilized in our testbed include the TORCS Racing Simulator [163], and CARLA autonomous vehicle simulator [179]. CARLA can be run on Windows, and Linux, but for our setup we have the simulator running on Ubuntu 18.04. A socket based communication is provided to access variables in the simulation, but we built a customized python API interface for easing variable access from external processes in the other distributed hardware in our testbed. The simulator can be customized to output sensor data such as lidar, speed, images, distance to objects, orientation, and GPS locations. Among the outputs, the user can change variables such as steering, acceleration, and braking. CARLA is the most sophisticated autonomous vehicle simulator to date and allows for us to develop more realistic experiments. TORCS can be run on Windows, Linux, and Mac computers, but for our setup we have the simulator running on Ubuntu 18.04. A socket based communication is provided to access variables in the simulation, but we built a customized python API interface for easing variable access from external processes in the other distributed hardware in our testbed. The simulator can be customized to output sensor values such as lidar, speed, brake, gear, track position, distance from start position, vehicle heading, and position in the race. Among the outputs, the user can change variables such as steering, acceleration, braking, and gear value.

CPS Controller: The software for the controller exists on the NVIDIA Jetson TX2 board. This board is configured with the Linux4Tegra 28.2 operating system, GPU libraries such as CUDA, and machine learning libraries such has Tensorflow. The operating system is additionally patched with the RT-PREEMPT patch. Furthermore, buffer overflow vulnerabilities are inserted to test the effect of a non-control data attack on the overall system behavior.

Communication: To support automotive applications, multiple communication interfaces are included such as Ethernet and CAN bus. For Ethernet communication, the ZeroMQ (ZMQ) communication library is utilized. Additionally, for the CAN bus communication, an open source library called SOCKETCAN is utilized to support the communication between the control code and ECU cluster.

IV.7 Case Study

For evaluation purposes, an autonomous vehicle case study is utilized to demonstrate the capabilities of our developed security architecture. It is important to note that our security architecture can be applied to any distributed CPS scenario utilizing underlying software computation processes, not just automotive scenarios.

IV.7.1 Scenario 1: Adaptive Cruise Control

This scenario is based on an adaptive cruise control scenario, with one manual vehicle driving as the leader and an autonomous vehicle representing the follower. For the purpose of evaluation, the follower vehicle will be the center of focus from a security perspective. The automotive system is comprised of electronic control units controlling steering, and throttle actuation, while receiving lidar, speed and orientation sensor readings as input. A PID controller is utilized to control driving behavior based on these inputs and outputs. However, our security architecture is generic meaning that it is not just limited to PID controllers, and any other controller software process can be utilized instead. The goal for this scenario is to control the follower vehicle to maintain a consistent distance behind the leader vehicle, as well as staying as close as possible to the center of the road.

IV.7.1.1 Attack Scenario

The follower vehicle is comprised of several components including a sensor and actuator ECU cluster, driving controller, telematics control unit (TCU), remote function actuator, and RFID sensor. There are two external interfaces including cellular communications from the TCU for remote monitoring services, and a RFID sync with the vehicle key fob. The driving controller constantly polls for the key fob signal to determine if the engine should remain on. When the key fob is within a close distance, the vehicle will be able to drive, but as soon as the key fob is out of communication range the vehicle will turn off. Under normal operation there is not a communication channel for the TCU to transmit input to the driving controller. However, since the TCU is connected remotely through a cellular interface, this component is the most at risk for being compromised by the adversary. Even though the attacker can't compromise the driving controller directly through the TCU, they can still utilize an intermediary step through the remote function actuator to inject malicious input into the driving controller. As such, the attack process consists of the following steps: 1) Compromise TCU through cellular communications, 2) Pivot to remote function actuator component, 3) Transmit malicious input to driving controller, 4) Overwrite target steering control value for PID controller. By utilizing the above process, the adversary can cause the follower vehicle to drive off the road, resulting in massive damage.

IV.7.2 Scenario 2: Advanced Emergency Braking System

This scenario is based on an Advanced Emergency Braking System (AEBS) scenario, with one manual vehicle driving as the leader and an autonomous vehicle representing the follower. For the purpose of evaluation, the follower vehicle will be the center of focus from a security perspective. The automotive system is comprised of electronic control units controlling braking and throttle actuation, while receiving lidar, and speed sensor readings as input. A Neural Network is utilized for the AEBS component of the driving controller, controlling the braking of the vehicle to avoid collisions with the leader. The AEBS Neural Network is a 3 layer sequential model created using the Tensorflow Lite library. During the case study, the leader vehicle will brake at a stop light, thus requiring the follower vehicle AEBS system to be activated. The goal for this scenario is for the AEBS controller to brake the follower vehicle to avoid colliding in the back of the stopped leader vehicle.

IV.7.2.1 Attack Scenario

The follower vehicle is comprised of several components including a sensor and actuator ECU cluster, driving controller, telematics control unit (TCU), remote function actuator, and RFID sensor. There are two external interfaces including cellular communications from the TCU for remote monitoring services, and a RFID sync with the vehicle key fob. The driving controller constantly polls for the key fob signal to determine if the engine should remain on. When the key fob is within a close distance, the vehicle will be able to drive, but as soon as the key fob is out of communication range the vehicle will turn off. Under normal operation there is not a communication channel for the TCU to transmit input to the driving controller. However, since the TCU is connected remotely through a cellular interface, this component is the most at risk for being compromised by the adversary. Even though the attacker can't compromise the driving controller directly through the TCU, they can still utilize an intermediary step through the remote function actuator to inject malicious input into the driving controller. As such, the attack process consists of the following steps: 1) Compromise TCU through cellular communications, 2) Pivot to remote function actuator component, 3) Transmit malicious input to driving controller, 4) Overwrite distance values for AEBS controller. By utilizing the above process, the adversary can cause the follower vehicle to collide into the back of the leader vehicle, resulting in significant damage to both vehicles.

IV.7.3 Results

IV.7.3.1 Static Analysis Performance

For the purpose of the case study, a PID controller is utilized for Scenario 1, while a 3 layer sequential Neural Network is utilized for the AEBS component of the driving controller in Scenario 2. From an implementation standpoint, there are 15 variables with a file size of 50 Kilobytes in Scenario 1, while there are 1025 variables with a file size of approximately 220 Kilobytes in Scenario 2. Additionally, there are three shared libraries that we need to secure in Scenario 2: tensorflow lite, libm, and ZeroMQ. This provides a good baseline for measuring the scalability of the different components in our static analysis pipeline.

The first stage in our DSR static analysis pipeline is binary lifting. In order to explore the variable space of our target program, it is necessary for the binary to be converted to LLVM bitcode. Mcsema is an increasingly popular tool that accomplishes this task. In our implementation Mcsema relies on Binary Ninja for the disassembly and control flow graph generation, and a custom developed python script to perform the instruction translation process. As such, it is important to note that the performance of the first step especially is variable dependent on the external disassembly tool that is utilized. The execution time of the Mcsema custom section is pretty consistent averaging approximately 15 milliseconds for Scenario 1, and 17 milliseconds for scenario 2, both of which were analyzed with 1000 executions. This conveys that there is relatively good scalability, and the execution times are satisfactory for our purposes since it is only necessary to perform this step once before load time.

The second stage in our DSR static analysis approach is points to analysis. In our implementation we use an open source implementation of the Andersen algorithm which provides polynomial time efficiency due to the context and flow insensitive approach. To evaluate the scalability of the points to analysis implementation we ran 100 iterations of generating PAGs for both the PID controller in Scenario 1 and Neural Network controller in Scenario 2. It was observed that the average execution time was approximately 120 milliseconds for Scenario 1 and 250 milliseconds for scenario 2. Even with this increased overhead, the execution times are reasonable and are only necessary once during program runtime.

IV.7.3.2 Experiment Results

IV.7.3.2.1 Scenario 1: Adaptive Cruise Control

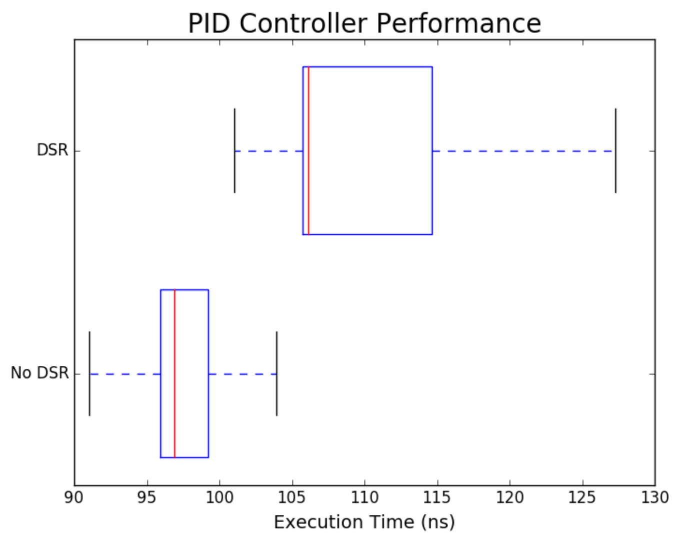


Figure IV.10: Controller Execution Times

Due to the target sampling rate of 20 Hz, it is paramount to limit the overhead of our security architecture. The overhead created with DSR enabled is minimal. For example, when looking at the PID controller execution times, overhead is about 10%, bringing the average execution time from approximately 96 microseconds to 106 microseconds. Additionally, this overhead brings the worst case execution time from 105 nanoseconds to 128 nanoseconds. These results represent the lower bound of our architecture overhead. Even with a scaling factor of 10, this is still well under the 50 millisecond deadline defined in the design process. However, it is important to note that the PID controller is a relatively small sized program. With a large sized program, there is potential for the overhead to increase significantly.

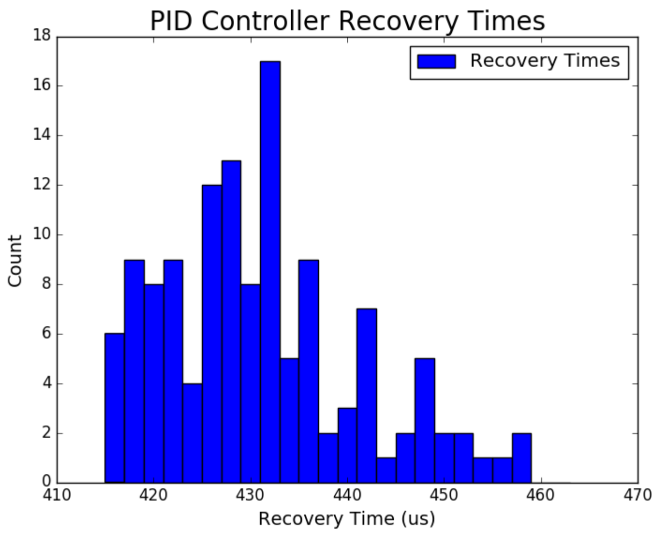


Figure IV.11: Recovery Times

During the case of a non-control data attack, the adversary is able to manipulate the PID controller operation by altering the target steering angle. With this adjustment, the new target steering value is set at the value of -1, causing the vehicle to make a hard left turn. Due to the fact that this value is constantly transmitted to the vehicle controller from the remote function actuator, the vehicle remains in a constant turning state and consequently performs donuts until crashing into the wall. However, in the scenario where DSR and reconfiguration is activated, the attack attempt will fail, and control will be transferred to a backup PID controller with a new randomization environment, decreasing the probability of a successful future attack. Furthermore, when looking at the damage of the follower vehicle behind the leader vehicle, the resultant attack disrupts the platoon behavior causing the following vehicle to be left behind in a crashed state. With reconfiguration enabled, the following vehicle continues to have reliable and safe operation. The vehicle driving behavior can be observed in Figure IV.12.

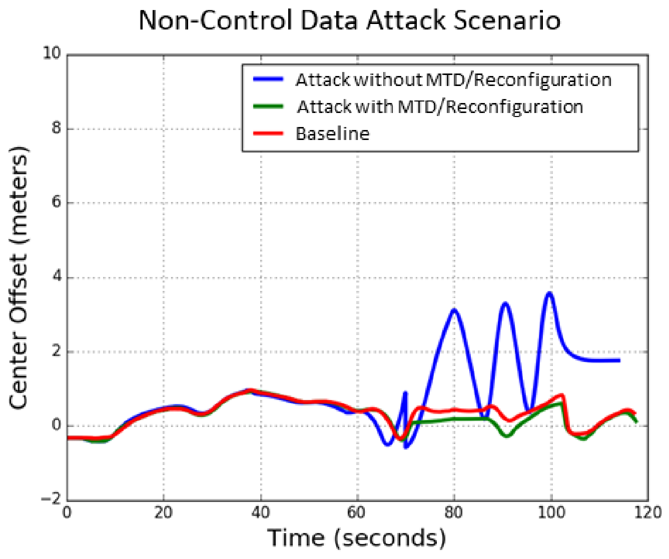


Figure IV.12: Road Center Offset Time Plot

IV.7.3.2.2 Scenario 2: Advanced Emergency Braking System

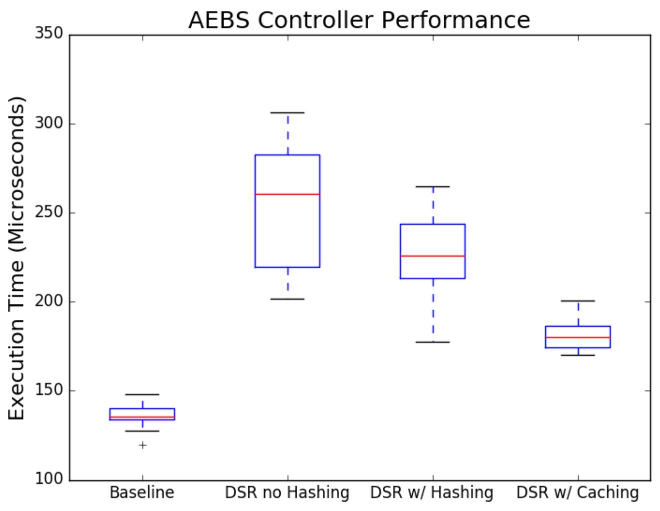


Figure IV.13: Controller Execution Times

Due to the target sampling rate of 20 Hz, it is paramount to limit the overhead of our security architecture. As shown in Figure IV.13, the overhead created with DSR enabled is pretty significant. To measure the overhead we analyzed the execution times for 1000 iterations of our DSR approach with varying inputs. When looking at the AEBS controller execution times, overhead is approximately 83%, bringing the average execution time from approximately 135 microseconds for the baseline scenario to 247 microseconds with our DSR implementation enabled. Additionally, this overhead brings the worst case execution time from

187 microseconds to 321 microseconds. Most of the overhead presented from our approach arises from accessing the randomization key lookup tables. To improve the overhead, we created a slight change within the lookup table to utilize hashing. We use a simple modulo approach with 100 slots to store randomization keys based on the memory location address that the instruction is accessing. With this approach, we were able to decrease our overhead from 85% to 61%, a 27% reduction. The average execution time with hashing is approximately 217 microseconds, while the worst case execution time was 274 microseconds. Finally, to further improve the performance overhead we implement caching for the variable encryption keys. As such, instead of constantly accessing the lookup table, at the first instruction instance, we store the encryption key adjacent to the load/store instruction in program memory space. Therefore, every subsequent time this instruction is encountered will require only accessing the adjacent encryption key with an added instruction instead of utilizing program handlers to access the variable lookup table. For the first 100 iterations, this approach produces a performance overhead consistent with the hashing table approach, ranging around 60%. However, after the 100 iteration mark when a majority of the load and store instruction encryption keys were cached, we were able to further decrease our overhead to an average of approximately 34%, allowing for a more reasonable amount of overhead for real time applications.

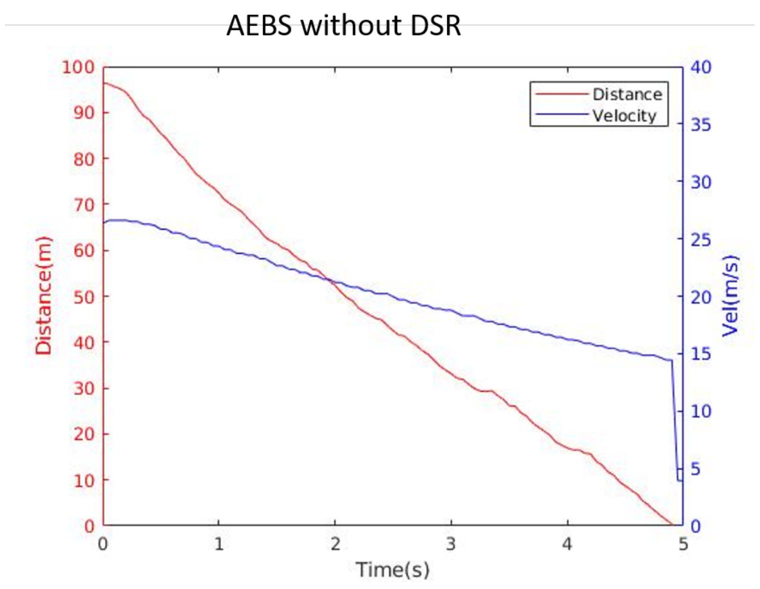


Figure IV.14: Non-Control Data Attack without DSR

During the case of a non-control data attack, the adversary is able to manipulate the AEBS controller operation by altering the perceived distance to the leader vehicle. With this adjustment, the new distance value is set at the max value of 100 meters, causing the follower vehicle to maintain acceleration. Furthermore, this continuance of acceleration combined with the rapid approach towards the leader vehicle will result in the crashing between the two vehicles as illustrated in Figure IV.14. However, with DSR and variable integrity checking enabled, the attempt by the attacker to overwrite the distance variable will result in an incorrect variable comparison, consequently flagging the attack. At this point, a fail safe controller will take over execution and fully brake the vehicle. As a result, safety will be preserved and the follower vehicle will avoid colliding with the stopped leader vehicle as observed in Figure IV.15.

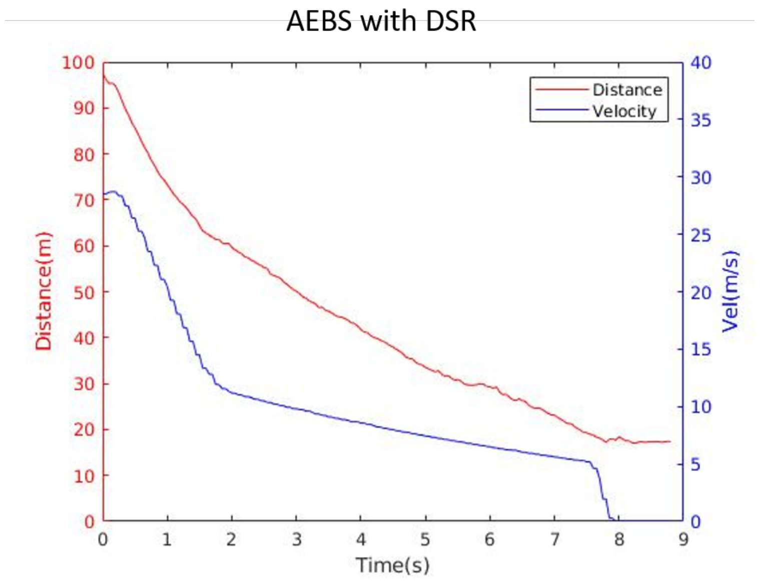


Figure IV.15: Non-Control Data Attack with DSR

IV.8 Conclusion

In this chapter we have shown how DSR can be integrated with variable integrity checking techniques in autonomous vehicles for the purpose of ensuring secure, and reliable operation. Due to the tightly coupled nature between the cyber and physical components in CPS, it is not just acceptable to maintain data integrity, but is necessary to guarantee a safe state of operation. Furthermore, non-control data attacks are a viable technique for altering physical behavior without the need for manipulating program control flow. Instead of overwriting the function return address on the stack, these attacks overwrite adjacent data variables to the input buffer with the goal of utilization in safety-critical operations. DSR can protect against non-control data attacks by changing the representation of variables, leaving adversary reconnaissance obsolete. As such, any manipulation of data variables will be vastly different compared to the intended goal. Furthermore, by including a duplicate stored variable with a different randomization key, a comparison can be performed at variable load time to detect the presence of variable tampering. Our approach was tested with a hardware in the loop testbed and an autonomous vehicle case study with an AEBS controller to illustrate CPS behavior on embedded hardware similar to deployment environments. By performing experimentation we found that our framework produced positive security protection against non-control data attacks and limited physical behavior effects, while introducing reasonable performance overhead to the system. In the next chapter, we plan to tie all of work together by integrating our DSR security approach with ISR, and ASR to include protections against code injection and code reuse attacks in addition to non-control data attacks.

CHAPTER V

Implementation of Moving Target Defense in Mixed Time and Event Triggered Cyber-Physical Systems

V.1 Introduction

Cyber-Physical Systems (CPS) often are designed for safety-critical applications, with a necessity for producing computation and actuation output in a predictable manner, providing for safe, and consistent operation of the controlled physical processes. As such, CPS such as medical devices, automobiles, and military applications are considered real time systems, requiring that underlying computation processes execute within stringent time constraints. These applications are further considered hard real time systems meaning that any failure to execute a given computation process by the respective deadline results in a complete failure of the system. With these requirements it is significant to have reliable real time scheduling mechanisms for ensuring that system tasks consistently meet their respective deadline constraints.

Time triggered scheduling is a commonly utilized technique for hard real time CPS applications. In time triggered scheduling techniques, underlying tasks are scheduled solely on the tick of a clock, avoiding the use of any other aperiodic event such as receiving sensor input or detecting a fault. As such, tasks execute at a predetermined point in time that is repeated continuously. For instance, at design time, a static schedule is normally created organizing every operating task into an execution block of a larger time period. Task execution blocks are then organized in the static schedule in a contiguous manner to develop a super period for the entire schedule. This period has to satisfy time constraints necessary for maintaining stability in utilized control algorithms, as well as minimizing the amount of time taken for detecting faults through polling tasks.

Code injection, code reuse, and non-control datay attacks pose a serious threat to present day CPS. Utilizing any one of these attack vectors allow adversaries to remotely control and alter sensor data, providing the capability to execute unsafe actuation behavior, while making it look like normal functionality to the monitoring employees at the central operating station. Instruction set randomization (ISR), address space randomization (ASR), and data space randomization (DSR) are all very effective moving target defense techniques (MTD) that mitigate these types of attacks. However, when disrupting the attack process, these techniques result in the software crashing due to an exception such as an invalid instruction, invalid address, or unsafe data value, thus increasing the risk of denial of service behavior in the system. However, in CPS, system availability and the integrity of the static schedule is absolutely critical, making this behavior unacceptable.

In CPS, timing is a critical component to maintaining safe behavior. A majority of software consists of real time constraints that are relied upon for the integrity of scheduling processes. Regardless of hard or soft real time constraints, a failure in meeting deadlines can result in dangerous consequences. Therefore, recovery mechanisms must be put in place to ensure execution transfer to a backup process while reconfiguring the static schedule to avoid any missed deadlines.

Safety-critical CPS often utilize time triggered architectures in order to guarantee predictable and reliable operation. However, in the event of a cyber-attack, it is necessary to include the rapid response benefits of event triggered architectures instead of waiting until the following period of the static schedule to respond. The main problem that arises in this chapter is how do we combine time and event triggered architectures in order to provide predictable, and reliable operation during normal circumstances, but also include rapid detection and reconfiguration capabilities in the event of a cyber-attack occurrence. In our approach we protect against code injection, code reuse, and non-control data attacks by developing a mixed time and event triggered MTD security architecture. Further, we consider how to limit the overhead of our approach on the designed static schedule configurations. Finally, we focus on how to reconfigure our system during an attack to limit the amount of missed deadlines. The hypothesis in this paper is that by integrating ISR, ASR, and DSR with the ARINC 653 standard, we can protect against code injection, code reuse, and non-control data attacks while rapidly reconfiguring in time to maintain system safety during cyber-attacks. The contributions of this chapter are as follows:

- We develop a mixed time and event triggered MTD security architecture that includes predictable and reliable operation during normal circumstances, while including rapid detection and reconfiguration during cyber-attacks. Our security architecture includes ISR, ASR, and DSR implementations to protect against code injection, code reuse, and non-control data attacks. Furthermore, we leverage the benefits of the ARINC 653 architecture for the time triggered design in our approach.

- We implement a reconfiguration scheme to detect and recover from cyber-attacks rapidly enough to limit missed deadlines and maintain safe and reliable operation.

- We present an autonomous vehicle case study to demonstrate the effectiveness of our security architecture in limiting the impact of cyber-attacks in the context of an advanced emergency braking system (AEBS) scenario. Furthermore, implement our architecture using a hardware-in-the-loop testbed to evaluate the benefits on hardware consistent with deployment environments.

V.2 Background

V.2.1 ARINC 653

ARINC 653 is a popular standard implemented within aerospace and other safety-critical CPS applications for the purpose of maintaining safety and predictability within critical components [152]. As this standard forms the foundation behind our approach in the rest of the paper, it is important to describe the basic components to this architecture. ARINC 653 is comprised of two types of components: partitions, and processes. These components have various types of attributes, states, and communication methods that can be configured throughout the design process. These components are described below.

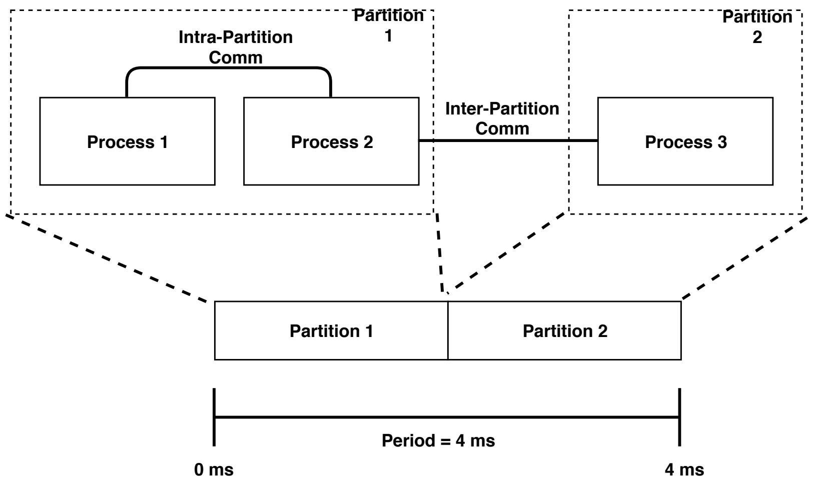


Figure V.1: Time Triggered Application Architecture

V.2.1.1 Partitions

Partitions are the highest level of abstraction in the ARINC 653 standard and include a shared memory space running tasks. Partitions can essentially be considered the same as a virtual machine in a single application environment comprising its own data, context, and attributes. Additionally, due to the segmented memory spaces between partitions including temporal and spatial activity separation, fault effects can be isolated and the safety of the overall system can be maintained. Furthermore, each partition has a fixed amount of memory and temporal duration. The following characteristics of partitions are described below.

V.2.1.1.1 Attributes

Partitions are made up of several attributes that can be configured to control the finest details of execution. The various fixed partition attributes include:

- Identifier - used to access partition from other parts of the application

- Memory Usage - memory bounds with code/data segregation. Memory allocation is fixed.
- Period - Amount of time given for partition execution in the static schedule
- Duration - CPU time given to partition for given period instance
- Criticality - How important partition is in overall context of application
- Communication - Methods and receiving actors that partition communicats with
- Health Monitor Table - Used to monitor behavior of underlying tasks in partition
- Entry Point - used to start and restart address

There are also other optional partition attributes that can be defined including a lock level, operating mode, and start condition. The start condition can be used to define specific preconditions that can trigger the partition while operating mode can define the specific state of operation within the partition.

The most critical of these attributes include the identifier, memory usage, and period. The identifier allows the partition to be referenced by other portions of the application, while the memory usage attributed is utilized to reference the upperbound memory usage for the respective partition. It is important to note that the memory allocation is fixed, so this value must be verified to not be exceeded. Finally, in order to create a static schedule for the application, each partition must be denoted with a period in the context of the whole super frame. This period denotes how much time is allocated to the specific partition so designers much ensure that this period encompasses enough time for all included tasks to finish their executions within the partition. If the tasks are not able to finish execution, they will be paused until the next partition instance, meaning that output values could be missed.

V.2.1.1.2 Modes

At any given time, a partition can be defined with one of several possible modes. A mode controls the behavior of the respective partition, such as whether tasks can execute, how long it will take for the partition to start up, and whether memory can be accessed. The possible partition modes are described below:

- Idle - The partition is not executing any processes
- Normal - The partition is executing processes
- Cold Start - Initialization phase is in progress and partition is executing initialization code. Can't interact with hardware interrupts
- Warm Start - Initialization phase is in progress with ability to interact with hardware interrupts.

The modes that are most relied upon are the idle and normal modes. In the idle mode, even though the partition is allocated for a specific time window for operation, the process scheduler will not be active and the underlying processes will not run. However, in normal mode, all processes have the capability to operate in the respective time window. By switching between these two modes, designers can replicate event triggered control to define constraints for specific operations within the application.

V.2.1.2 Processes

In a partition, there can exist multiple processes that correspond to the actual tasks in an application. It is easiest to think of the partition in terms of a virtual machine, and a process as the specific software applications running inside of the virtual machine. As such, multiple processes can run concurrently and are reliant on the operating system for their respective scheduling capabilities. Similarly to regular applications, processes can be initialized, terminated, reinitialized, and paused through control commands sent by the parent partition.

V.2.1.2.1 Attributes

Besides the executable itself, a process also contains other data structures such as a stack and data area, program counter, stack pointer, priority index, and deadline. Additionally, several other attributes can be defined at design time. These attributes are listed below:

- Name - Unique ID for each process
- Entry Point - Starting address for the executable
- Stack Size - Overall size of the runtime stack
- Base Priority - Criticality of the process
- Period - Time allotment for activation within a partition
- Time Capacity - Elapsed time in which processes should complete execution
- Deadline - Type of deadline
- Process State - Scheduling State of the Process

V.2.1.2.2 States

Each process can be idetnified at any given time with one of four possible states. These states include dormant, ready, running, and waiting. A dormant states represents when a process is not able to receive resources or otherwise turned off such as before it is started or after it is terminated. A ready state is when the process is able to be executed but is waiting to be scheduled by the operating system. A running state is when the process is currently running on the processor. Finally, a waiting state is when the process is not allowed to execute until a specific event occurs.

V.2.1.3 Communication

Communication within the ARINC 653 framework is conducted through the use of unidirectional channels, ports and messages. Additionally, communication is possible both between partitions (inter-partition), as well as within a partition between multiple processes (intra-partition). It is important to note that the data transmitted is considered an atomic entity meaning that if the whole messages isn't received than no part of the message will be received. As such, it is important at design time to ensure that the communication and message structures are setup properly.

The fundamental data structure for communication is called a message. Each message transmits pre-fixed amounts of data from a source component to a destination component. Additionally, messages can be specified as different types such as periodic, aperiodic, broadcast, or unicast, and different respective sizes of data.

Messages are transferred from the source to destination through a channel. Additionally, each channel is connected to the respective source and destination components through ports. Ports can be defined as either a single memory allotment (sampling port) or queue like structure.

V.2.1.3.1 Inter-Partition Communication

For inter-partition communication, channels can be utilized to transfer messages between partitions within an application. As such, each partition will have the option to include one of two types of ports: a sampling port, or queuing port. A sampling port is a single memory allotment to store received data. As such, whenever a message is received, the existing data stored in the port will be overwritten by the new data accordingly. This type of port is best utilized for sensor streaming where the goal is to fetch the most updated data value.

The second type of message port is referred to as a queuing port. In contrast to a sampling port where the message data is overwritten, queuing ports implement a queue data structure that keeps track of every message that is received. As such, the destination partitions can be triggered by certain types of messages, or can implement logic that can take into account data from multiple messages. This type of message port is best for partitions that need to keep track of a history of messages.

V.2.1.3.2 Intra-Partition Communication

For intra-partition communication, channels can be utilized to transfer messages between processes within partitions. Each process has the option to use two types of communication: buffers, or blackboards. For buffers, each new message has the possibility to carry new data and thus cannot overwrite previous received messages. This type of communication is most similar to the queuing mode for inter-partition communication. Blackboards on the other hand, do not have the capability to keep track of multiple received messages. Blackboards are similar to sampling ports in that they can only have one memory allotment for incoming data. When a new message is received, the original data will be overwritten by the new incoming data.

V.2.1.4 Scheduling

At the backbone of the ARINC 653 standard is the temporal schedule. The temporal schedule, also referred to as a static scheduled in real time systems, is a sequence of consecutive partition time allotments that build up to create one large periodic time allotment. As such, the larger time allotment which is referred to as a superframe period will continuously repeat itself, resulting in a repetition of the same partition sequence over and over again. The partitions themselves are constrained to only execute within their specific time allotment within the overall superframe. As such, at any given time this schedule is deterministic, improving the reliability of the overall system. This partition period is referred to in the ARINC 653 context as a hyper period.

It is important to note that the partitions themselves are scheduled in a static manner, processes within partitions don't necessarily have to execute every iteration. It is also possible for processes in partitions to remain idle until triggered by a specific event. As such, this behavior can allow designers to replicate event triggered behavior and create sporadic components within the larger time triggered architecture.

V.2.2 Moving Target Defense Techniques

As defined in Chapter II, the fundamental MTD techniques that form the backbone of our work include ISR, ASR, and DSR. Legacy CPS software often contains numerous memory corruption vulnerabilities such as buffer overflows that allow attackers to remotely perform code injection, code reuse, and non-control data attacks. By randomizing the various internal structure of software, attacker reconnaissance efforts are ineffective, resulting in failed cyber-attack attempts.

In a code injection attack, adversaries are able to leverage a buffer overflow vulnerability to inject and execute code remotely on the program stack. To successfully execute code, the inject instructions format must be consistent with the native system processor format (x86, ARM, etc.). By randomizing the representation of native instructions at runtime with ISR, any attacker injected code will be of an invalid representation, resulting in an invalid instruction exception.

In a code reuse attack, adversaries are able to leverage a buffer overflow vulnerability to divert control flow to another location within program memory such as a safety-critical function. To be successful, attackers must have knowledge of the memory location of their target in order to divert program control flow effectively.

By randomizing the address layout with ASR, the location of target functions will no longer be as expected by the attacker, resulting in diverting control flow to the wrong location. As such, any code reuse attack attempts will result in an invalid memory access exception.

In a non-control data attack, adversaries are able to leverage a buffer overflow vulnerability to overwrite adjacent safety-critical variables. To be successful, attackers must have knowledge of the variable format, allowing them to correctly alter the variable value. With DSR, the variable representations will be randomized, resulting in any attacker manipulation to result in a value wildly different then expected. This increases the ease in detecting any malicious data tampering activity, preventing the attacker from successfully overwriting critical variables.

V.2.3 Control Reconfiguration

As defined in Chapter II, it is not just enough to stop cyber-attacks, but CPS necessitate the requirement to maintain safe operation as well. As such, availability is a key CPS property that must be maintained in the event of a cyber-attack. The most popular approach to control reconfiguration within safety-critical CPS applications is the simplex architecture [107]. Simplex also forms the backbone of our developed security architecture. Simplex is comprised of two controllers: a default controller for normal execution, and a safety controller that serves as a backup in case of failure in the default controller. The default controller is designed to be high performance, while potentially containing vulnerabilities. The safety controller on the other hand is not designed to be as optimal from a performance standpoint, but guarantees safe, and secure operation. Additionally, Simplex includes a decision module that determines the circumstances when to switch between the two controllers, as well as actually performing the execution transitioning process. By utilizing this approach, safety-critical CPS can include the benefits of high performance during normal operation while safely maintaining operation during a fault or cyber-attack.

V.3 Problem Formulation

In safety critical CPS, there is often a tradeoff between predictability and flexibility. Advances in computing technology have allowed for a more digitized environment in safety-critical systems such as automobiles. The significant increase in computing power combined with faster and higher throughput communication buses has now transformed the modern day automobile into essentially a computer on wheels. With this rapid transformation, it is necessary to ensure that the critical predictability properties remain in tact, as any failure in correct system behavior can result in significant damage to the surrounding environment. In order to maintain predictability, the modern day automobile is considered a real time system, meaning that each computation task must be completed by a predetermined deadline in order to ensure that each actuation event executes at a rapid enough frequency to support safe operation.

The two types of real time scheduling approaches include event triggered and time triggered scheduling. In event triggered architectures, the system activities, such as sending a message or starting computational activities, are triggered by the occurrence of events in the environment while in time triggered architectures activities are triggered by a sequence of global time [145]. As such, event triggered architectures are controlled through interactions with stimuli in the environment, while time triggered architectures are autonomously controlled through a static predefined schedule of events, supporting autonomous control flow. Time triggered architectures are advantageous in that at any given point of time, the system user will always be confident on what process is executing, providing a degree of predictability in the system operation behavior. As such, systems can be fully verified with 100% certainty that no corner cases will occur resulting in system failures. This makes time triggered architectures the approach of choice for safety-critical CPS, which necessitate a high degree of reliability [146; 147]. For soft real time systems in which safety is not a direct result of predictable operation, event triggered architectures are preferred due to the higher degree of flexibility and more efficient performance.

A majority of safety-critical CPS designs incorporate a time triggered architecture in order to guarantee predictable and reliable operation during all circumstances. However, in the event of a cyber-attack, traditional time triggered architectures would have to incorporate polling mechanisms in order to detect the attack occurrence. This means that at worst case, it would take a whole period for an attack to be detected, seriously jeopardizing the safety and integrity of executing processes. In order to improve the response time of attack detection and recovery, event triggered functionality must be integrated to ensure that the system can recover from an attack fast enough to minimize missed deadlines and maximize system reliability. Furthermore, it is also important for the recovery process to preserve the time triggered predictability properties going forward. As such, this chapter focuses on creating a mixed time and event triggered architecture for maintaining the predictability of operation while including rapid detection and response during the course of a cyber-attack.

Finally, safety-critical CPS often contain memory corruption vulnerabilities such as buffer overflows that allow for the remote exploitation of software. Cyber-attacks such as code injection, code reuse, and non-control data attacks allow for adversaries to hijack safety-critical functionality, potentially resulting in severe damage to the surrounding environment. The main necessity to successfully perform any of the above attacks is reconnaissance knowledge such as the instruction architecture, location of critical functions, and variable representations. By integrating MTD such as ISR, ASR, and DSR into our architecture, we can disrupt the reconnaissance process, making any subsequent attacker exploit invalid. As such, the rest of the chapter focuses on a developed mixed time and event triggered MTD security architecture that is able to maintain predictable operation during normal circumstances, while rapidly and reliably protecting, detecting, and reconfiguring from code injection, code reuse, and non-control data attacks.

V.4 Threat Model

An exemplary vehicle system model includes 6 components: a sensor cluster, actuator cluster, driving controller, telematics control unit (TCU), remote function actuator (RFA), and RFID sensor. The sensor cluster provides critical data representing the current state of the vehicle such as the speed, and front facing camera image. The actuator cluster provides the ability to manipulate the vehicular acceleration through the throttle and brake, and vehicle angle through steering. The driving controller is responsible for performing computation based on the provided sensor cluster input, and outputting commands to the actuation cluster. In this paper, the driving controller is an AEBS controller that is responsible for braking the vehicle to avoid colliding with upcoming objects. Both the TCU, and RFA are responsible for providing the external interface for the vehicle. The TCU monitors the various metrics of the system, transmitting data to a remote operating station for maintenance and emergency purposes. The RFA is responsible for determining the presence of a key fob for allowing the vehicle to be turned on.

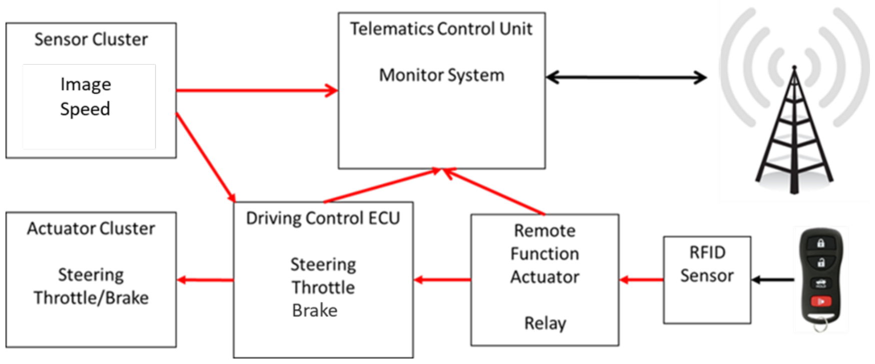


Figure V.2: Case Study

In the system model, the sensor cluster, actuator cluster, and driving controller are on a safety-critical CAN bus network, including both communication authentication to prevent spoofing, and integrity checking within the driving controller to ensure that utilized sensor data is accurate. On the other hand, the TCU, and RFA communicate with the driving controller through a low priority CAN bus interface. Since these components are the most vulnerable to remote attacks due to being connected to external communication channels, the safety-critical and low priority communication buses protect against the TCU and RFA directly controlling the sensor or actuator ECU clusters. However, to detect the presence of the key fob, the driving controller constantly polls for status updates from the RFA. This communication is authenticated to prevent message spoofing, but there is a memory corruption (buffer overflow) vulnerability in the driving controller that provides an opportunity for memory corruption attacks.

The attack model for this chapter focuses on a code injection, code reuse, and non-control data attack on a vehicle network. The authors in [159] note that the biggest current threat to self driving vehicles is exploitation through remote avenues. As such, the attack vector utilized in this paper consists of the adversary compromising the TCU through the remote cellular interface, and consequently pivoting to hijack the RFA. With access to a direct communication channel with the driving controller, the adversary can craft a message payload to take advantage of the memory corruption vulnerability and alter control. At this point, an attacker can perform an attack where they can leverage the buffer overflow to effect safety-critical behavior in the driving controller.

It is important to note that due to the one way communication constraint enforced by the ARINC 653 standard [152], it is only possible for the attacker to compromise the partition containing the vulnerable controller. Therefore, it is impossible for the attacker to compromise any of the other partitions in the static schedule including sensor collection, and perception. As such, only the processes in the driving controller partition will include MTD defense protections in our approach.

Five assumptions are made for our approach to be successful. First, it is assumed that the sensor and actuator clusters are fully secure. The driving controller ECU contains the buffer overflow vulnerability utilized for control hijacking, while the TCU and RFA contain vulnerabilities allowing for key fob message spoofing. Second, the attacker has knowledge of the relative address of a safety-critical variable relative to the start of the input buffer. Third, the attacker has knowledge of the underlying software architecture of the safety-critical controllers, allowing them to target the most impactful variables. Fourth, our static analysis approach has to overapproximate the points-to set for the driving controller instructions. Finally, the driving controller binary has the variable data types stored internally, allowing for accurate variable type recovery at the intermediate level. These assumptions are not impractical given examples demonstrated in the literature [31].

To evaluate the effectiveness of our architecture within the context of an autonomous vehicle case study, we utilize a developed hardware-in-the-loop testbed. We further utilize physical metrics such as vehicle position combined with software metrics like performance overhead in both normal operation and attack scenarios. Finally, to conclude that our hypothesis is true two observations need to be clear from the results: 1) The performance overhead needs to be minimal enough to ensure that execution times do not exceed designed real time constraints and 2) Vehicles need to follow safe driving behavior, stopping completely before colliding with the parked vehicle on the road. In the event that both of these observations are true, we can conclude that our architecture is successful.

V.5 Architecture

V.5.1 Architecture Components

The key components in our architecture are the (1) CPS Controllers which control the physical plant, (2) DBT which uniquely customizes the runtime environment for each CPS controller, (3) points to analysis graph (PAG) which describes the relationship between pointers and variables within a program, and (4) Module Manager which controls the reconfiguration of our system during an attack.

V.5.1.1 CPS Controller:

This component is the actual software that controls the CPS application. From the most generic form, the controller takes sensor input from the system, performs computation operations, and outputs actuation commands to perform in the surrounding physical environment. Our architecture incorporates a generalist approach, allowing for a broad array of control techniques and applications to be supported. The only requirement is that the control program be in the form of an Elf binary. As such, development is performed with C++ versus higher level languages such as Python.

V.5.1.2 Dynamic Binary Translator (DBT):

This component is responsible for providing a unique randomization backend for each spawned CPS controller in the architecture. In other words, the DBT is a virtual sandbox layer that serves as an intermediary between the executing binary and the processor. The DBT has the ability to intercept instructions as they are fetched and alter program semantics before execution by the processor. As such, a DSR methodology is supported by encoding variables before storage on the stack, as well as a derandomizaton stage before loading into registers. Each variable is supported to include a dynamically generated unique randomization key, as well as a duplicate comparison variable for detecting variable tampering. Additionally, the DBT is responsible for storing a variable key mapping table which incorporates uniquely generated keys for every variable in a program. The open source instrumentation tool Mambo is utilized to support the DBT implementation within our architecture.

V.5.1.3 Points to Analysis Graph:

This component is responsible for identifying the relationship between variables within a program. By feeding this information into our DSR implementation, we can identify correct randomization keys to utilize for respective memory locations.

V.5.1.4 Module Manager:

This component is responsible for recognizing an attack attempt, and rapidly reconfiguring the system to spawn backup controllers to take over functionality and minimize safety-critical component downtime. Additionally, the Module Manager is responsible for executing the static schedule for time triggered functionality.

We make the assumption that the CPS controller component in our architecture is vulnerable to cyber-attacks by the adversary. The remaining components are not susceptible to cyber-attacks. As such, the variable key storage table in each DBT is assumed to be secure against integrity attacks in our threat model. Our security architecture is designed with the goal of keeping the CPS controller from becoming compromised by the attacker.

V.5.2 Design Time

V.5.2.1 Time Triggered Design

In order to design our time triggered architecture, we utilize the ARINC 653 standard. ARINC 653 allows us to distribute our program into separate, isolated partitions executing in a sequential order. Each partition includes its own memory space, meaning that if a process executing in another partition crashes, all other partitions will not be effected. In our design, we first have to perform an execution analysis on all of the relevant system components, identifying the maximum time required for the processes to complete. After this step, we build in slack time, and assign the respective required time allotments for each partition in the system. It is important to note, that the assigned time allotment must be larger than the maximum execution time of the underlying process. Otherwise, system processes may not fully finish, and the behavior of the system could be consequentially effected.

After the integration of our MTD architecture, it is important to note that there is a degree of overhead that exists. As such, original deadlines may no longer be feasible. Therefore, it is necessary to factor in MTD overhead into the design of the time triggered static schedule.

V.5.2.2 Moving Target Defense

In order to secure our most critical applications against code injection, code reuse, and non-control data attacks, it is necessary to implement a MTD wrapper. As demonstrated in our previous work, utilizing ISR, ASR, and DSR can mitigate against potential attacks while reducing the impact of reconnaissance efforts on our system. Our updated architecture is now integrated with all three MTD techniques to produce the most comprehensive defense benefits. In the rest of this section we will discuss our MTD architecture as well as the steps necessary for configuration.

The fundamental basis behind our architecture is the impact of MTD approaches. In order for an attacker to be successfully, they usually need to a large amount of reconnaissance effort to discover key system properties such as instruction architecture, function locations, and data representation. MTD takes advantage of these key requirements by dynamically adjusting key system properties to make any reconnaissance knowledge gained by the attacker obsolete. The three main MTD approaches utilized in our architecture are discussed below.

In our ISR implementation, we randomize instructions with a 32 bit key dynamically generated at runtime, creating a high degree of entropy protection for an attacker to correctly guess the new instruction architecture. For our ASR implementation, we shuffle functions and randomize base memory addresses, significantly decreasing the probability of success of code reuse and return oriented programming attacks. Finally, for our DSR implementation we XOR stack based variables with a 64 bit key, while also utilizing a redundancy comparison check for determining the presence of a non-control data attack.

Both ISR and ASR have different degrees of granularity that you can use to optimize the tradeoff between security and performance. ISR can use two types of randomization: either XOR or AES 256 encryption. Furthermore, we also built in the capability to randomize different memory ranges with different keys to reduce the likelihood of the adversary correctly guessing the randomization key. ASR by default is defined with course grained granularity meaning that the base addresses of the program, stack, heap, and shared libraries are different for every runtime instance. However, we also built in the capability for fine grained granularity meaning that not only are the base addresses unique but funtion locations are shuffled as well.

V.5.2.3 Lift Target Binary

For the purpose of static analysis, it is optimal to convert the binary program into an intermediate representation (IR) format. The low level virtual machine (LLVM) compiler includes an IR representation called LLVM bitcode which is the most widely utilized representation for this purpose [169]. Since many existing static analysis tools already use this IR, it makes sense to convert the binary program to this representation. This process is defined as binary lifting. To convert a native binary to LLVM bitcode two steps are necessary: control flow recovery, and instruction translation [171]. Control flow recovery includes analyzing the execution path of a program for the purpose of understanding the specifics of how a program functions. In the academic community it is standard to rely on an existing tool such as IDA Pro [173] or Binary Ninja [172] to compute the control flow graph due to the state of the art features and significant past work put in place. We continue this trend by relying on Binary Ninja for control flow recovery in our approach.

V.5.2.4 Points-To-Analysis

In a program, an object can either be an instruction or a memory location. For DSR to be successful, it is important to have context of the relationships between instructions and respective memory locations. Points-To-Analysis is utilized to produce the associations between instructions and the memory locations that they access through loads and stores. By having knowledge of the unique memory objects we can define unique randomization keys for each location. Additionally, by keeping track of the associations between the memory locations and instructions, we can create a map corresponding to what randomization keys to utilize for respective instructions in the program. As such, during the DSR implementation, every time we encounter that respective instruction we will know what key to use for the randomization or derandomization process. The Points-To-Analysis process produces a Points-To-Analysis Graph (PAG) as output which allows for visually, and iteratively identifying the relationships between instructions and memory locations.

V.5.3 Runtime

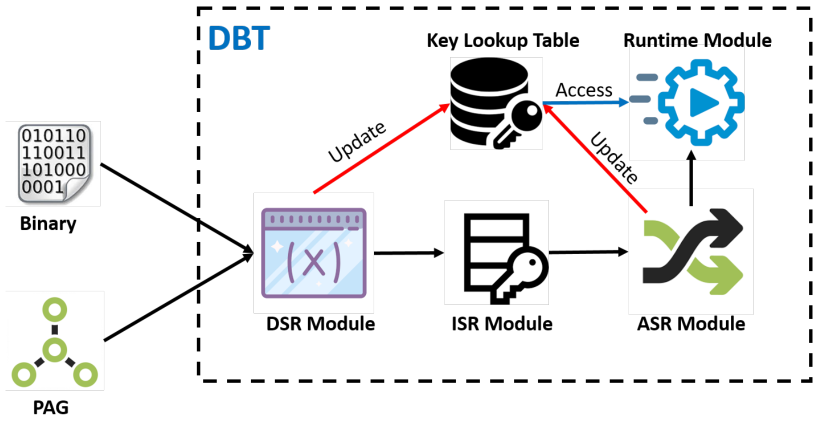


Figure V.3: MTD Initialization Process

The runtime environment for our MTD approach is integrated into the DBT component encompassing the vulnerable CPS controller. However, there are several steps that are taken for our runtime to become initialized once the configuration steps are followed. After the last configuration which is the PAG integrity checking phase, there will be two inputs to our DBT: the binary executable itself, and a text file defining the instruction and memory associations from the PAG. Once these inputs are received by the DBT, the binary will begin to be pushed through the randomization pipeline. First, the binary will be pushed throug hthe DBT module. In this step a key lookup table will be generated by parsing the input PAG text file to store the respective randomization key to utilize for each load/store instruction. Next, the binary instructions will be randomized through the ISR module. During this process, a 32 bit key will be dynamically generated and each instruction in the program will be iteratively randomized through an XOR operation. After this step, the binary will be push through the ASR module where the functions will be shuffled. Since after this step, the program instructions will have different addresses compared with during the DSR stage, we adjust the DSR table with the updated addresses of the respective load/store instructions. Finally, we start execution of the newly randomized binary. After every instruction is fetched, it will be derandomized by performing another XOR operation before the being decoded by the pipeline. Additionally, anytime there is a store instruction a randomization instruction will be inserted which XORs the value with the respective associated key from the lookup table and stores the variable at the appropriate location on the stack. Additionally, a duplicate copy with a duplicate randomization key will also be stored in an adjacent location to the newly randomized variable. When a load instruction encountered, both copies of the variable will be loaded and derandomized with their respective keys found from the lookup table. After derandomization, the plain text values are compared for equivalence. If both values are equal then the program can proceed as normal. However, in the event that the values are different, an "Attacked Variable" exception is generated and the program is terminated.

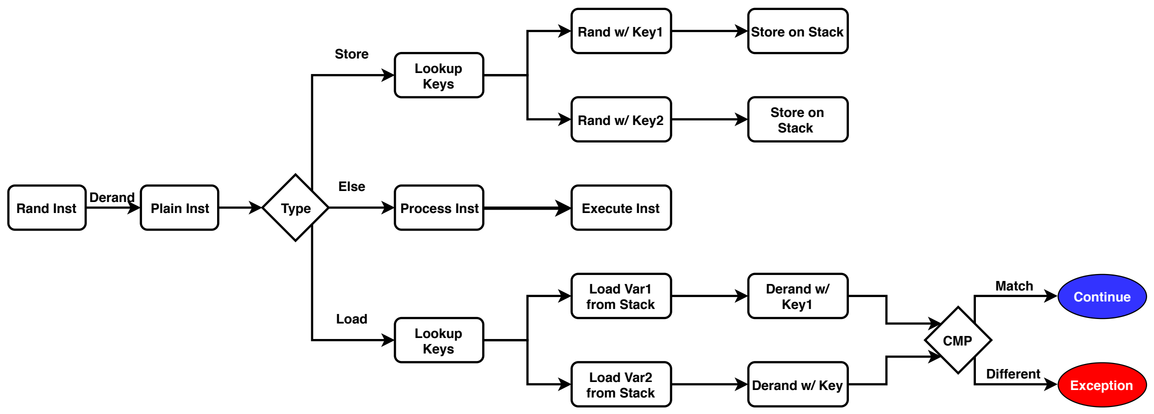


Figure V.4: MTD Runtime Instruction Pipeline

V.5.4 Control Reconfiguration

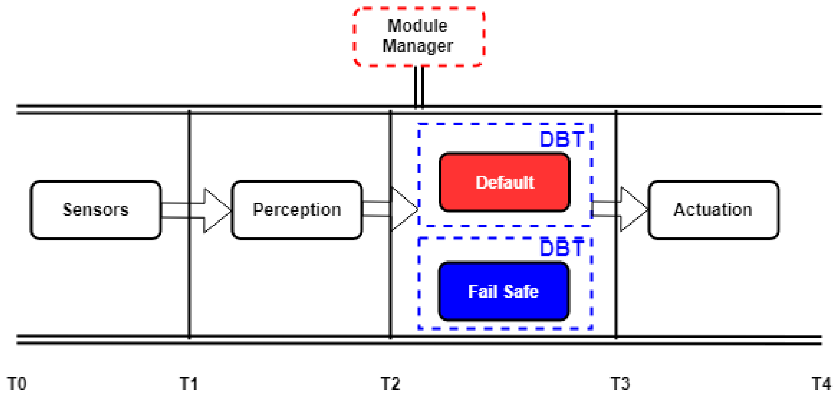


Figure V.5: Reconfiguration Setup

In safety-critical CPS, it is not just enough to stop a cyber-attack, but it is equally as important to maintain availability and safe operation of physical actuation. In time triggered applications it is also important to maintain the integrity and determinism of the static schedule, ensuring that the overall system is predictable and reliable. Therefore, it is important for reconfiguration capabilities to ensure that even if an attack occurs, we can transfer execution to backup controllers while minimizing the amount of deadlines that we miss. This responsibility is given to the Module Manager component in our architecture which focuses on detecting cyber-attack instances and then handling the reconfiguration control. When an attack is attempted and thwarted through MTD protections, the respective process will be terminated through an invalid instruction, invalid address access, or attacked variable exception. Once termination occurs, a the Module Manager detects the exception and transmits a signal to the backup controller process to resume execution.

V.6 Implementation

To be successfully implemented, our approach must include a combination of three components including time triggered capabilities, randomization capabilities, and reconfiguration capabilities. These are described in further detail below.

V.6.1 Time Triggered Functionality

To be more supportive to a majority of safety-critical CPS, it is critical to transition from a strictly event triggered approach to mixed time triggered and event triggered functionality. As such we make use of the ARINC 653 standards for designing our integrated architecture. To support rapid development of our implementation, we chose to use an ARINC 653 software emulator [180]. This decision allows us to ease the transition from the traditional Linux environment to design a static schedule, as well as keeping the ability to utilize POSIX signals and event triggered functionality. In our autonomous vehicle case study we create partition to represent specific categories of functionality within the system (sensors, perception, computation, actuation), while specifying tasks as the underlying tasks executing operations. Furthermore, we utilize sampling ports to communicate between partitions be updating the respective data appropriately between partitions. This includes safety messages that monitor the health status of critical software components within our system.

V.6.2 Moving Target Defense

To support randomization within our architecture, we utilize DBTs which are developed utilizing MAMBO [56] within our implementation. MAMBO is a dynamic binary translation tool designed for ARM systems that allows for manipulating and monitoring binaries at runtime for the purpose of dynamic analysis. However, for our purposes MAMBO is beneficial in allowing for implementing our randomization algorithms dynamically to avoid leaking any key security data while stored within the file system. MAMBO includes an ELF loader stage that allows for us to randomize and shuffle program instructions as they are stored from the file system to runtime memory. Furthermore, MAMBO includes a virtual runtime pipeline that includes several stages such as fetch, decode, edit, and execute. This capability is huge for allowing us to access and manipulate instructions as they are fetched before they reach the processor for execution. In this stage, we are able to derandomize the instructions after being fetched but before being decoded, while additionally randomizing and derandomizing variale data for respective load and store instructions before being sent to the processor for execution.

To support DSR in our implementation we must have a reliable static analysis capability. Static analysis in our security framework is achieved through 1) variable identification, and 2) points-to analysis. For variable identification, iteration is performed over the Elf symbol table to access the full spectrum of variables within the target program. Afterwards, points-to-analysis is performed utilizing a combination of Mcsema [170], and SVF [176]. Mcsema is a binary lifter that utilizes a combination of control flow recovery and instruction translation to convert the program executable to an intermediate code representation that is compatible with current points-to analysis algorithms in the LLVM compiler [169]. Due to the vast amount of research put into solutions such as Binary Ninja [172], it makes sense to utilize these solutions for binary disassembly and control flow graph recovery versus developing a full disassembler from scratch. With the outputted control flow graph, Mcsema includes an open source instruction translation implementation to convert the underlying disassembly instructions into LLVM intermediate bitcode. After the intermediate bitcode is obtained Anderson points to analysis is performed with the open source SVF library. SVF incorporates three steps consisting of graph, rules, and solver that takes a program control flow graph as input, formulates a set of variable constraints based on the Anderson methodology, and applies these constraints throughout the program utilizing a constraint solver in the LLVM compiler environment. After this stage, a GraphViz representation of the pointer and variable associations will be outputted in a dot file.

V.6.3 Control Reconfiguration

The final segment of our implementation is the reconfiguration capability. To properly reconfigure we base our implementation on the Simplex architecture [107]. Simplex is a fault tolerant methodology that prioritizes safety and availability of a CPS. As such, the architecture is based on four main components: sensors, computation, a decision module, and a physical plant which is controlled through actuation. The computation is comprised of two sub components representing a high performance controller, and safety controller. The high performance controller is more optimal for normal use but may have vulnerabilities and faults that could lead to problems down the line. However, the safety controller is less optimal in terms of performance but is generally designed to be guaranteed to be fault and vulnerability free. As such, when a cyber-attack occurs, the safety controller is the most reliable option for use until the system is later stabilized. The decision module is responsible for controlling which one of the controllers executes within the system. As such, the decision module usually includes predetermined algorithms defining when and how to transfer execution between controllers. In our implementation for our autonomous vehicle scenario, the high performance controller is a reinforcement learning driving controller, while the safety controller is a fail safe controller that automatically brakes the vehicle before any dangerous collisions occur. Additionally, the decision module is represented by the Configuration Manager within our architecture. Our implementation will start with the default driving controller while once an attack is detected the configuration manager will switch execution to the backup safety braking controller.

V.6.4 Experiment Setup

To enhance the ability to evaluate cyber-attack impacts in deployment environments, a hardware in the loop testbed was developed. Our testbed includes embedded hardware representing CPS software infrastructure, a simulation workstation representing the physical environment, and multiple network interfaces representing communication channels within the automobile environment. The setup of our testbed includes four components: 2 beaglebone black microcontrollers [162] representing the sensor and actuator processes in an automobile ECU cluster, a NVIDIA Jetson TX2 board [161] providing for computational power necessary for designing vehicle control algorithms, an i7 simulation desktop, and a realtime web based results dashboard. Furthermore, two communication interfaces exist including 100 Mbps ethernet, and a 1 Mbps CAN bus. The hardware architecture is illustrated in Figure V.6.

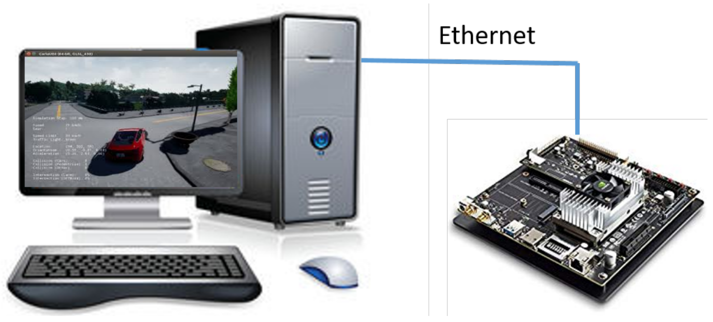


Figure V.6: Testbed Hardware Architecture

V.6.4.1 Software Architecture

The software architecture of the testbed provides the capability to implement real time CPS control algorithms to interact with and operate an autonomous car within a connected simulator.

Autonomous Vehicle Simulator: The autonomous vehicle simulator utilized in our testbed is the CARLA autonomous vehicle simulator [179]. CARLA can be run on Windows, and Linux, but for our setup we have the simulator running on Ubuntu 18.04. A socket based communication is provided to access variables in the simulation, but we built a customized python API interface for easing variable access from external processes in the other distributed hardware in our testbed. The simulator can be customized to output sensor data such as lidar, speed, images, distance to objects, orientation, and GPS locations. Among the outputs, the user can change variables such as steering, acceleration, and braking. CARLA is the most sophisticated autonomous vehicle simulator to date and allows for us to develop more realistic experiments.

CPS Controller: The software for the controller exists on the NVIDIA Jetson TX2 board. This board is configured with the Linux4Tegra 28.2 operating system, GPU libraries such as CUDA, and machine learning libraries such has Tensorflow. The operating system is additionally patched with the RT-PREEMPT patch. Furthermore, buffer overflow vulnerabilities are inserted to test the effect of a non-control data attack on the overall system behavior.

Communication: To support automotive applications, multiple communication interfaces are included such as Ethernet and CAN bus. For Ethernet communication, the ZeroMQ (ZMQ) communication library is utilized. Additionally, for the CAN bus communication, an open source library called SOCKETCAN is utilized to support the communication between the control code and ECU cluster.

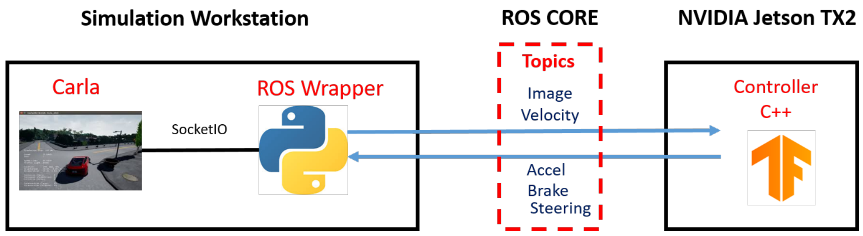


Figure V.7: Testbed Setup

V.7 Evaluation

V.7.1 Case Study

For evaluation purposes, an autonomous vehicle case study is utilized to demonstrate the capabilities of our developed security architecture. It is important to note that our security architecture can be applied to any distributed CPS scenario utilizing underlying software computation processes, not just automotive scenarios.

The case study is based on a platoon scenario, with one manual vehicle driving as the leader and an autonomous vehicle representing the follower. For the purpose of evaluation, the follower vehicle will be the center of focus from a security perspective. The automotive system is comprised of electronic control units controlling braking and throttle actuation, while receiving image, and speed sensor readings as input. A convolutional neural network is first utilized for perception to transform the image into a distance estimation to the leader vehicle. Then a driving controller utilizes a 3 layer sequential neural network for the braking component, and PID controller for throttle component to compute the optimal actuation of the car. Finally, a driving manager converts the output from the PID and neural network computation into an appropriate speed up or slow down decision to control the throttle and braking. Both the AEBS and perception neural networks were created using the Tensorflow Lite library. During the case study, the leader vehicle will brake at a stop light, thus requiring the follower vehicle AEBS system to be activated. The goal of this case study is for the AEBS controller to brake the follower vehicle to avoid colliding in the back of the stopped leader vehicle.

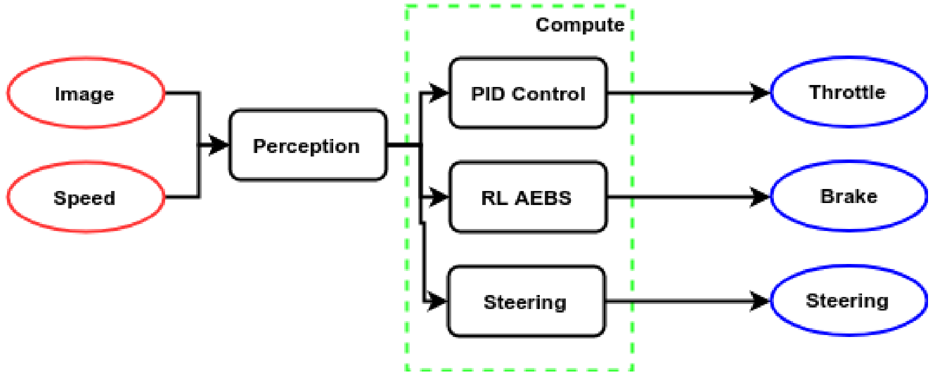


Figure V.8: Autonomous Vehicle Case Study

V.7.2 Attack Scenario

As illustrated in Figure V.2, the follower vehicle is comprised of several components including a sensor and actuator ECU cluster, driving controller, telematics control unit (TCU), remote function actuator, and RFID sensor. There are two external interfaces including cellular communications from the TCU for remote monitoring services, and a RFID sync with the vehicle key fob. The driving controller constantly polls for the key fob signal to determine if the engine should remain on. When the key fob is within a close distance, the vehicle will be able to drive, but as soon as the key fob is out of communication range the vehicle will turn off. Under normal operation there is not a communication channel for the TCU to transmit input to the driving controller. However, since the TCU is connected remotely through a cellular interface, this component is the most at risk for being compromised by the adversary. Even though the attacker can’t compromise the driving controller directly through the TCU, they can still utilize an intermediary step through the remote function actuator to inject malicious input into the driving controller.

Once the adversary injects malicious input into the driving controller, there are 3 attack scenarios that can occur. The first scenario revolves around a code injection attack where the attacker will inject a malicious payload to open a remote shell. At this point, the attacker disables the currently operating driving controller and starts a malicious controller to fully accelerate into the upcoming parked car. The second attack scenario consists of a code reuse attack in which the attacker utilizes the buffer overflow vulnerability to divert control flow to a steering function within the driving controller software. At this point the vehicle will turn left in a loop, resulting in a crash into a wall on the side of the road. The final attack scenario consists of a non-control data attack in which the attacker utilizes the buffer overflow vulnerability to overwrite a an adjacent safety critical variable within the AEBS component of the driving controller representing the distance to the leader parked vehicle. At this point the AEBS algorithm will believe that the leader vehicle is further ahead than reality, resulting in the follower vehicle crashing into the back of the parked car. These three scenarios provide a survey of the potential types of attacks based off of buffer overflow vulnerabilities within a safety-criticl CPS. It additionally illustrates the flexibility of our MTD approach to protect and recover from varying types of cyber attacks.

V.7.3 Time Triggered Setup

To appropriately utilize our architecture, it is necessary to develop a time triggered static schedule that turns the operations of our autonomous vehicle into a sequence of periodic sub-operations. In compliance with the ARINC 653 standards, we first need to split our vehicle operations into categories which will represent the partitions in our time triggered schedule. In terms of our case study, we will have 5 partitions representing sensor data receipt, state estimation, processing, decisions, and actuation transmission. The sensor partition is responsible for receiving the updated image, and speed data from the CARLA simulator ROS interface. The state estimation partition which contains a perception process then computes an estimated distance based off of the image forwarding the speed sensor data along with it. Then processing partition, which contains a computation process determines an optimal throttle and braking value based upon a neural network and PID algorithm. After this stage, the decision module which contains a driving manager process utilizes predefined logic to determine whether the vehicle should brake or speed up based upon the received requested throttle and braking values. Finally, the output command will be fed to the actuation partition which will transmit the requested command back to the CARLA simulator through the ROS interface. The structure of our partitions and processes can be observed in Figure V.9.

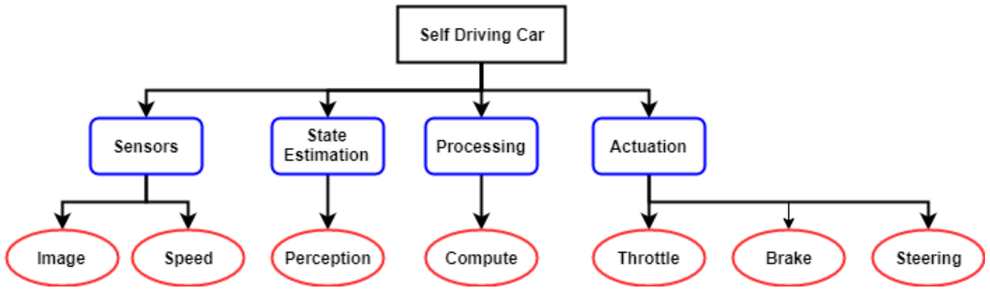


Figure V.9: Autonomous Vehicle Partition and Process Breakdown

Process Execution Times			
	Min	**Avg**	**Max**
Sensors	200 us	221 us	256 us
State Estimation	31.2 ms	45.6 ms	52.1 ms
Compute	118 us	138 us	160 us
Compute w/ Rand	182 us	218 us	258 us
Module Manager Recon.	320 us	467 us	489 us
Fail Safe	30 us	42 us	51 us
Actuation	183 us	209 us	231 us

Table V.1: Execution Time Analysis for Autonomous Vehicle Processes

V.7.3.1 Static Schedule

To accurately establish the static schedule for our case study, we must conduct an execution time analysis to determine the appropriate allocated amount of time for each partition. Too much allocated time can result in an inefficiency in the approach while too little allocated time can result in a failure in the system to operate correctly. To identify the upperbound of each respective process, we record 100,000 iterations under varying conditions. With this data we can identify the average, as well as outlier values that could potentially exceed the deadline established.

After performing the execution time analysis we identified the following statistics for each process identified in Table V.1. A more intensive analysis of the randomization overhead on the compute process is described in the results subsection.

After the maximum execution times for each process are found we need to ensure that we establish enough slack time in our deadline decisions to take into account more possible outliers and unexpected behaviors. Additionally, when taking into account the deadline to establish for the decision partition we must also factor in the possibility of the configuration manager and fail safe processes executing as well due to an attack. Therefore, we established the following deadlines assigned to the respective partitions in the case study: the sensors partition will be 10 ms, state estimation partition will be 60 ms, compute partition will be 10 ms, decision partition will be 10 ms, and actuation partition will be 10 ms.

As such, the static schedule utilized in this case study is illustrated in Figure V.10. The super frame period is defined as 100 ms and will repeat continuously throughout the systems lifetime. This period is small enough to support optimal and safe functionality of the vehicle, while being large enough to provide enough slack time to comfortably support system processes.

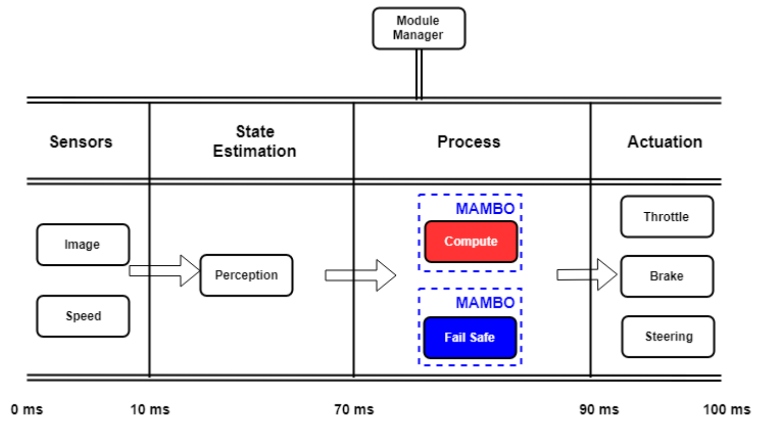


Figure V.10: Autonomous Vehicle Static Schedule

In order to verify that our static schedule is accurate, we need to analyze the worst case scenario by determining if the aperiodic Module Manager attack detection, and recovery processes fit within the critical slots of the Partition 3 schedule. If this case is shown to be true, then the designed schedule is guaranteed to be fully schedulable [148]. In our designed schedule the default controller exists in Partition 3 while the recovery processes and fail safe controller will be triggered by an attack. This means that at the earliest, the Module Manager attack detector will be triggered at the beginning of Partition 3 and at the latest at the end of Partition 3. Since the default controller is the only initial process in Partition 3, the Module Manager and fail safe controller processes will have full access to the CPU for the remainder of partition 3. However, as a worst case scenario, we take the case of the attack occurring at the end of the Partition 3 time allotment. In this case, fail safe controller won't finish execution until the a full period later during the next time allotment of Partition 3. During the missed execution, the previous actuation command will be utilized, providing a degree of stability to the vehicle behavior.

V.7.4 Results

V.7.4.1 Static Analysis Performance

For the purpose of the case study a 3 layer sequential Neural Network is utilized for the AEBS component of the driving controller. Additionally, there is a PID component responsible for speed control, as well as a steering controller responsible for keeping the vehicle on a straigh path. However, these two components are negligible in size compared to the AEBS learning enabled component, leaving most of our analysis focusing on the overhead presented by analyzing the AEBS component. From an implementation standpoint, there are 1025 variables with a file size of approximately 220 Kilobytes. Additionally, there are two shared libraries that we need to secure: tensorflow lite, and libm. This provides a good baseline for measuring the scalability of the different components in our static analysis pipeline.

ARINC 653 Configuration				
	Part1	**Part2**	**Part3**	**Part3**
# of Processes	1	1	2	1
Period	10 ms	60 ms	20 ms	10 ms
Base Priority	99	99	99	99
Stack Size	4096	4096	4096	4096
Deadline Type	Soft	Soft	Soft	Soft

Table V.2: ARINC 653 Experiment Configuration

The first stage in our DSR static analysis pipeline is binary lifting. In order to explore the variable space of our target program, it is necessary for the binary to be converted to LLVM bitcode. Mcsema is an increasingly popular tool that accomplishes this task. In our implementation Mcsema relies on Binary Ninja for the disassembly and control flow graph generation, and a custom developed python script to perform the instruction translation process. As such, it is important to note that the performance of the first step especially is variable dependent on the external disassembly tool that is utilized. For neural network program it appears that the execution time of the Mcsema custom section is pretty consistent averaging approximately 17 milliseconds for 1000 executions. This conveys that there is relatively good scalability, and the execution times are satisfactory for our purposes since it is only necessary to perform this step once before load time.

The second stage in our DSR static analysis approach is points to analysis. In our implementation we use an open source implementation of the Andersen algorithm which provides polynomial time efficiency due to the context and flow insensitive approach. To evaluate the scalability of the points to analysis implementation we ran 100 iterations of generating PAGs for neural network controller. It was observed that the average execution time was approximately 250 milliseconds. Even with this increased overhead, the execution times are reasonable and are only necessary once during program runtime.

V.7.4.2 Experiment Results

For the purpose of replicating the experiment results, that ARINC 653 configuration for our setup is specified in Table V.2. It is important to note that the total period of the super frame equates to 100 ms which is a realistic value for modern day autonomous vehicles. Furthermore, the stack size is the default value for the ARINC 653 implementation which is 4096. Additionally, both the sensor value receipt and actuation transmission capabilities have been combined into one process in their respective partition instead of segmenting each sensor and actuation value into separate processes. This is why Partition 1 contains 1 process instead of 2, and Partition 4 contains 1 process instead of 3. Finally, the deadline types for each partition have been set to a soft real time constraint, meaning that if the deadline is missed, the process will finish at the next partition instance. This is in contrast to the hard real time constraint option in which case would terminate when a deadline is missed.

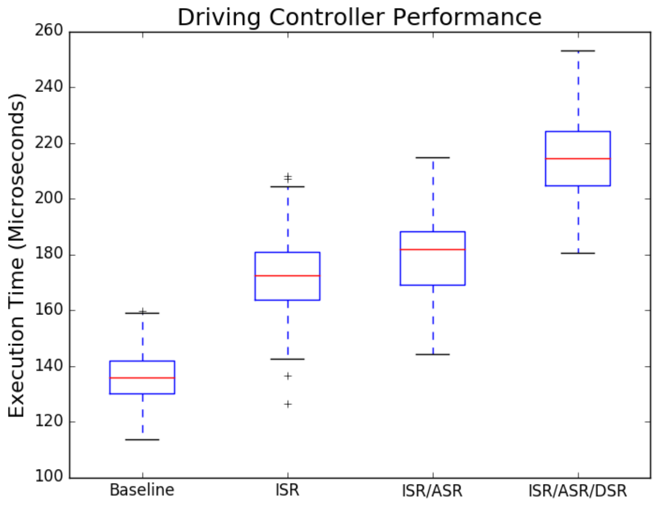


Figure V.11: Driving Controller Execution Times

Due to the target sampling rate of 10 Hz, it is paramount to limit the overhead of our security architecture. As shown in Figure V.11, the overhead created varies depending on enabling a combination of ISR, ASR, and DSR. DSR has the largest impact on overhead at 33%, while ISR follows at approximately 14%. Due to only being executed at load time, ASR has a negligible effect at aroudn 1-2%. In our scenarios, we have three combinations enabled including one with only ISR, a second with ISR and ASR, and a final combination with all three MTD techniques enabled. With only ISR enabled, there is an overhead of approximately 14% while with ISR and ASR there is a little higher overhead at approximately 16%. However, with the significant increase in overhead created by DSR, the final combination with all three techniques enables results in an overhead at approximately 42%. As such, in cases where significant overhead is unacceptable, performing a risk assessment is necessary to conclude the most optimal combination choices.

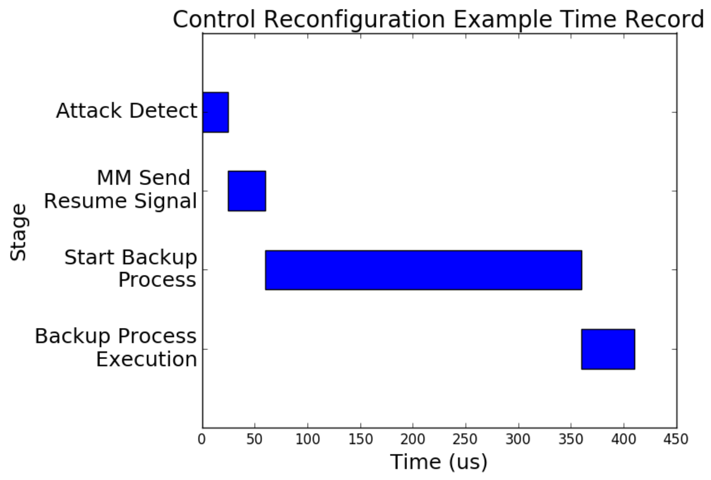


Figure V.12: Control Reconfiguration Example Time Record

Figure V.12 illustrates the time record of an example reconfiguration process. As such, the recovery process consists of four possible stages: attack detection, Module Manager send resume POSIX signal, start Backup Controller process, and execute Backup Controller to compute a new actuation value. As observed, the attack detection, and POSIX signal stages are relatively negligible while the controller start and execution time consume most of the reconfiguration time. Therefore, the reconfiguration time from attack detection to transmission of a new actuation command directly correlates to the size and execution time of the backup controller.

V.7.4.2.1 Code Injection Attack Scenario

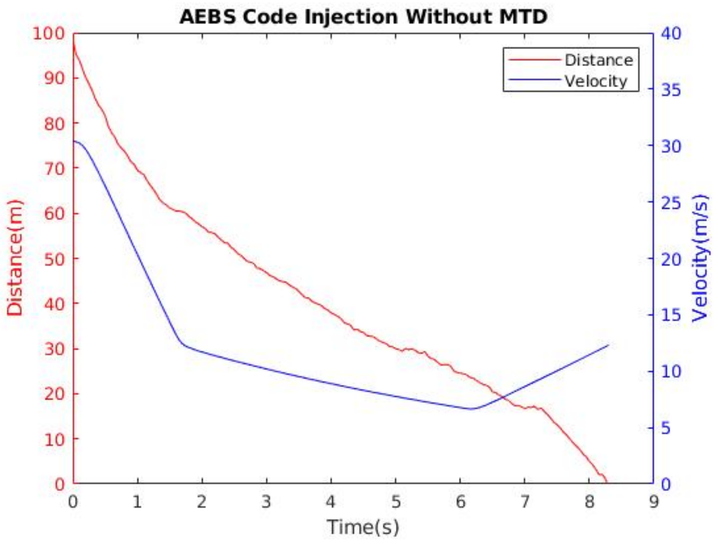


Figure V.13: Code Injection Attack without ISR and Recovery

During the case of a code injection attack, the adversary is able to inject a malicious payload to spawn a remotely accessible root shell within the vehicle operating system. At this point, the possibilities are limitless for the attacker to reek havoc. However, for demonstration purposes. in this scenario the attacker will terminate the default driving controller and spawn a malicious controller that will fully accelerate the vehicle in a straight path. At this point, as can be observed in Fiugre V.13, the vehicle will speed up and crash into the back of the parked leader vehicle, inflicting massive damage through a dangerous high speed collision. However, with ISR enabled in our MTD approach, the instruction architecture of the driving controller will become unique due to the execution within the DBT. This means that the code injection attempt will result in an inaccurate instruction formal, resulting in an invalid instruction execution exception and successful recover to a backup controller which fully brakes the vehicle. As such, Figure V.14 shows that we were able to successfully recover to the backup failsafe controller in time to brake the vehicle before crashing into the back of the leader parked car.

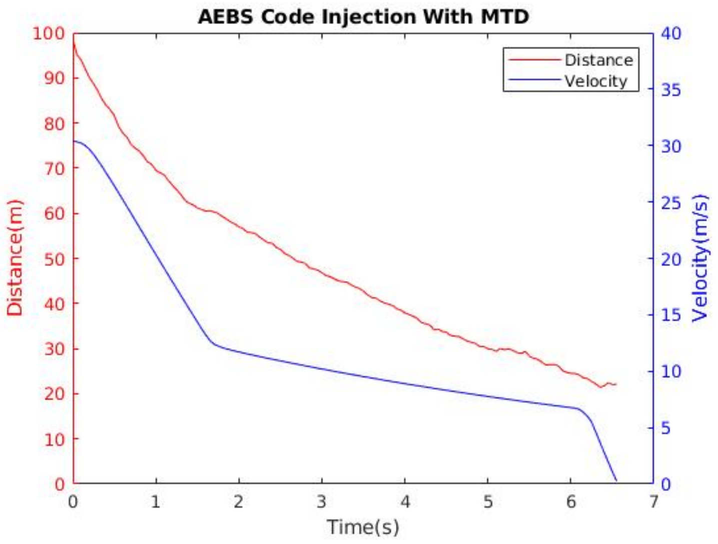


Figure V.14: Code Injection Attack with ISR and Recovery

V.7.4.2.2 Code Reuse Attack Scenario

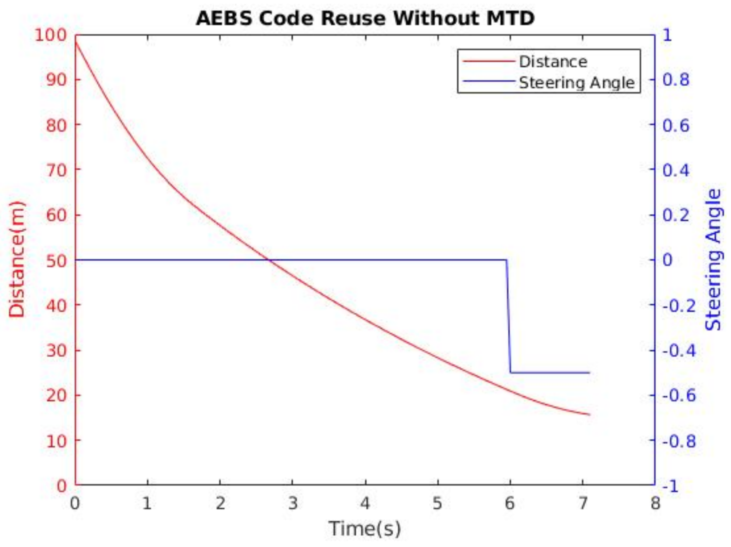


Figure V.15: Code Reuse Attack without ASR and Recovery

During the case of a code reuse attack, the attacker leverages an existing function within the program instead of directly executing injected remote code. In this scenario, the attacker leverages a buffer overflow vulnerability to redirect control flow to an existing turn left function within the steering component of the driving controller. At this point, control flow will then execute the turn left function in a loop, resulting in the vehicle constantly turning left in a loop. As such, Figure V.15 illustrates that the attacker is able to successful divert the vehicle to turn left and drive off of the road, eventually crashing into a wall off of the side of the road. However, with ASR enabled in our MTD approach, the memory layout of the software will become different, including the order of functions within the program. As such, when the attacker tries to divert control flow to the turn left function, the function will no longer exist in the target memory address, consequently resulting in an invalid memory access exception. At this point, as illustrated in Figure V.16, we are able to successfully recover to the backup failsafe controller where we are able to fully brake the car before unsafe behavior occurs. As such, the car will remain on the road, and will avoid colliding with any objects including the leader vehicle or objects along the side of the road.

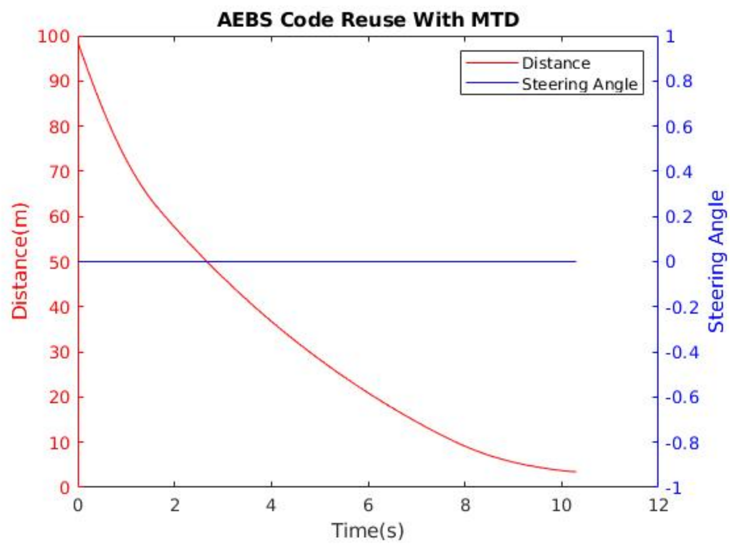


Figure V.16: Code Reuse Attack with ASR and Recovery

V.7.4.2.3 Non-Control Data Attack Scenario

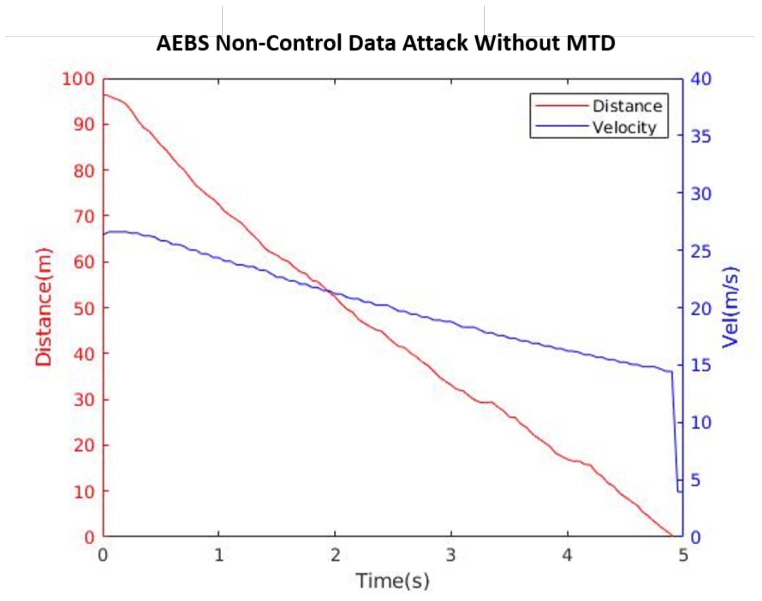


Figure V.17: Non-Control Data Attack without DSR

During the case of a non-control data attack, the adversary is able to manipulate the AEBS controller operation by altering the perceived distance to the leader vehicle. With this adjustment, the new distance value is set at the max value of 100 meters, causing the follower vehicle to maintain acceleration. Furthermore, this continuance of acceleration combined with the rapid approach towards the leader vehicle will result in the crashing between the two vehicles as illustrated in Figure V.17. However, with DSR and variable integrity checking enabled, the attempt by the attacker to overwrite the distance variable will result in an incorrect variable comparison, consequently flagging the attack. At this point, a fail safe controller will take over execution and fully brake the vehicle. As a result, safety will be preserved and the follower vehicle will avoid colliding with the stopped leader vehicle as observed in Figure V.18.

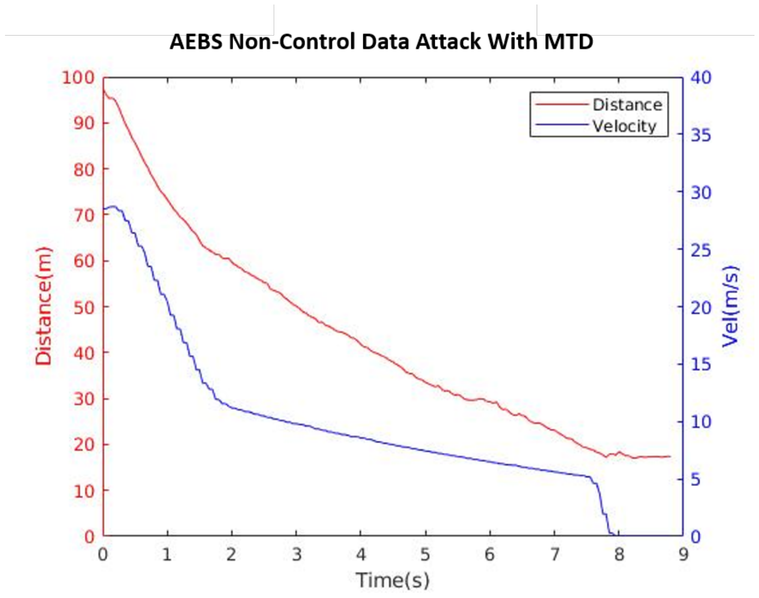


Figure V.18: Non-Control Data Attack with DSR

V.8 Conclusion

In this chapter we have shown how ISR, ASR, and DSR can be integrated to support protections against code injection, code reuse, and non-control data attacks in the context of safety-critical real time CPS applications. Our MTD architecture was successfully extended to a hybrid time triggered and event triggered architecture, and successful attack recovery is instrumented to transfer execution to backup fail safe controllers to avoid disruption in safety-critical operations. Furthermore, we are able to implement various levels of security to help designers balance the tradeoff between security and overhead. With ISR and ASR enabled we have a reasonable performance overhead while with the addition of DSR there is a significant overhead addition. Finally, by developing a hardware in the loop testbed to support our analysis of an automotive example, we are able to demonstrate our approach in a realistic setting similar to a deployment environment. By performing experimentation we foudn that our approach produced positive security protections against all three classes of attacks, while being able to recover to fail safe control in a rapid manner. By utilizing our MTD approach, we are able to transition security to a proactive preventative approach, creating CPS runtime environments that are resilient to buffer overflow based cyber-attacks.

CHAPTER VI

Conclusions

The increased growth and interest in the area of cybersecurity, especially as related to critical infrastructure, has creating many research problems in the CPS domain. It is no longer true that cyber attacks are limited to the information technology domain. Adversaries can now gain access to and manipulate dangerous devices such as military applications, autonomous vehicles, and medical devices that were once thought of to be safe. Additionally, it takes more than just securing the outer interfacing layer of a system with firewalls as once a zero day exploit gains access past the first layer, the adversary will have unrestricted access to the system. Due to this, there needs to be a larger emphasis on defense in depth techniques. Our developed security architecture aims to solve this problem by utilizing moving target defense techniques such as ISR, ASR, and DSR along with fault tolerance principles like diversity control structures to decrease the probability of an adversary successfully gaining entry into a system. Additionally, recovery mechanisms are integrated to ensure that once an attack attempt has been made the system will continue to execute reliably without losing availability. By leveraging moving target defenses and control reconfiguration, we have been successful in moving the field of cybersecurity forward and making our country a safer and more protected place.

CHAPTER VII

List of Publications

1. B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated Data Space Randomization and Control Reconfiguration for Securing Cyber-Physical Systems. In *Proceedings of the 2019 Symposium and Bootcamp on the Science of Security (HoTSoS)*, ACM, April 2019. (Best Paper Award)

2. B. Potteiger, H. Abdel-Aziz, H. Neema and X. Koutsoukos. Simulation Based Evaluation of Security and Resilience in Railway Infrastructure. In *Proceedings of the 2019 Symposium and Bootcamp on the Science of Security (HoTSoS)*, ACM, April 2019.

3. B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated Moving Target Defense and Control Reconfiguration for Securing Cyber-Physical Systems. In *Special Issue on Cyber-Physical Systems: Design and Applications*, Microprocessors and Microsystems, April 2019 (Pending).

4. B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated Instruction Set Randomization and Control Reconfiguration for Securing Cyber-Physical Systems. In *Proceedings of the 2018 Symposium and Bootcamp on the Science of Security (HoTSoS)*, ACM, April 2018.

5. H. Neema, B. Potteiger, X. Koutsoukos, G. Karsai, P. Volgyesi, and J. Sztipanovits. Integrated Simulation Testbed for Security and Resilience of CPS. In *Symposium on Applied Computing - Cyber-Physical Systems Track*, ACM , April 2018.

6. B. Potteiger, W. Emfinger, H. Neema, X. Koutsoukos, C. Tang, and K. Stouffer. Evaluating the Effects of Cyber-Attacks on CPS using a HIL Simulation Testbed. In *Resilience Week (RWS) 2017*, IEEE , September 2017. (Best Paper Award)

7. X. Koutsoukos, G. Karsai, A. Laszka, H. Neema, B. Potteiger, P. Volgyesi, Y. Vorobeychik, and J. Sztipanovits. SURE: A Modeling and Simulation Integration Platform for Evaluation of Secure and Resilient Cyber-Physical Systems. In *Proceedings of the IEEE Journal*, IEEE, 2017.

8. B. Potteiger, G. Martins, and X. Koutsoukos. Software and Attack centric integrated threat modeling for quantitative risk assessment. In *Proceedings of the 2016 Symposium and Bootcamp on the Science of Security (HoTSoS)*, ACM, April 2016.

9. A. Laszka, B. Potteiger, Y. Vorobeychik, S. Amin, and X. Koutsoukos. Vulnerability of transportation networks to traffic-signal tampering. In *International Conference on Cyber-Physical Systems*, April 2016.

10. H. Neema, P. Volgyesi, B. Potteiger, W. Emfinger, X. Koutsoukos, G. Karsai, Y. Vorobeychik, J. Sztipanovits. Demo Abstract: SURE: An Experimentation and Evaluation Testbed for CPS Security and Resilience In *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems(ICCPS)*, April 2016.

BIBLIOGRAPHY

[1] C. Smith, *The Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press, 2016.

[2] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security and Privacy*, 2014.

[3] B. Potteiger, Z. Zhang, and X. Koutsoukos, "Integrated instruction set randomization and control reconfiguration for securing cyber-physical systems," in *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, p. 5, ACM, 2018.

[4] "Gao-16-350, vehicle cybersecurity: Dot and industry have efforts under way, but dot needs to define its role in responding to a real-world attack." https://www.gao.gov/assets/680/676064.pdf. (Accessed on 02/14/2018).

[5] R. Baheti and H. Gill, "Cyber-physical systems," in *The impact of control technology* (T. Samad and A. Annaswamy, eds.), pp. 161–166, IEEE Control Systems Society, 2011.

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

[7] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces.," in *USENIX Security Symposium*, San Francisco, 2011.

[8] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, J. Baras, and S. Wang, "Toward a science of cyber–physical system integration," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 29–44, 2012.

[9] A. Cardenas, S. Amin, B. Sinopoli, A. Giani, A. Perrig, and S. Sastry, "Challenges for securing cyber physical systems," in *Workshop on future directions in cyber-physical systems security*, vol. 5, 2009.

[10] M. Wolf and D. Serpanos, "Safety and security in cyber-physical systems and internet-of-things systems," *Proceedings of the IEEE*, vol. 106, no. 1, pp. 9–20, 2018.

[11] X. KoutsouKos, G. Karsai, A. Laszka, H. Neema, B. Potteiger, P. Volgyesi, Y. Vorobeychik, and J. Sztipanovits, "Sure: A modeling and simulation integration platform for evaluation of secure and resilient cyber–physical systems," *Proceedings of the IEEE*, vol. 106, no. 1, pp. 93–112, 2018.

[12] N. Inkster, "Information Warfare and the US Presidential Election," *Survival*, vol. 58, pp. 23–32, sep 2016.

[13] S. Mukherjee, "Hackers Have Crippled Another Major Hospital Chain With a Cyberattack," 2016.

[14] P. Rosenzweig, "Alarming Trend of Cybersecurity Breaches and Failures in the U.S. Government," 2012.

[15] J. P. Farwell and R. Rohozinski, "Stuxnet and the Future of Cyber War," *Survival*, vol. 53, pp. 23–40, feb 2011.

[16] J. Riley, "Terrorism and Rail Security," 2004.

[17] D. Ortiz, B. Weatherford, M. Greenberg, and L. Ecola, "Improving the Safety and Security of Freight and Passenger Rail in Pennsylvania," tech. rep., RAND Corporation, 2008.

[18] R. Sanchez, "NJ train didn't have this safety system. Could it have stopped the crash? - CNN.com."

[19] C. Cerrudo, "Hacking US Traffic Control Systems," in *Defcon 22*, (Las Vegas, NV), 2014.

[20] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman, “Green Lights Forever: Analyzing the Security of Traffic Infrastructure,” 2014.

[21] H. Teso, “Aircraft Hacking Aircraft Hacking Practical Aero Series Practical Aero Series,”

[22] R. N. Charette, “This car runs on code,” *IEEE spectrum*, vol. 46, no. 3, p. 3, 2009.

[23] C. Meyers, S. Powers, and D. Faissol, “Taxonomies of cyber adversaries and attacks: a survey of incidents and approaches,” tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2009.

[24] T. Zhang and L. Delgrossi, *Vehicle safety communications: protocols, security, and privacy*, vol. 103. John Wiley & Sons, 2012.

[25] “Home — onstar.” https://www.onstar.com/us/en/home.html. (Accessed on 08/24/2017).

[26] “Lojack - lojack recovery system for cars, trucks, motorcycles, equipment, cargo & laptops.” http://www.lojack.com/. (Accessed on 08/24/2017).

[27] 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 *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pp. 1–12, IEEE, 2013.

[28] C. Miller and C. Valasek, “A survey of remote automotive attack surfaces,” *black hat USA*, vol. 2014, 2014.

[29] P. Carsten, T. R. Andel, M. Yampolskiy, and J. T. McDonald, “In-vehicle networks: Attacks, vulnerabilities, and proposed solutions,” in *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, p. 1, ACM, 2015.

[30] C. Miller and C. Valasek, “Adventures in automotive networks and control units,” *DEF CON*, vol. 21, pp. 260–264, 2013.

[31] C. Miller and C. Valasek, “Remote exploitation of an unaltered passenger vehicle,” *Black Hat USA*, vol. 2015, 2015.

[32] C.-W. Lin and A. Sangiovanni-Vincentelli, “Cyber-security for the controller area network (can) communication protocol,” in *Cyber Security (CyberSecurity), 2012 International Conference on*, pp. 1–7, IEEE, 2012.

[33] A. Perrig, R. Canetti, J. D. Tygar, and D. Song, “Efficient authentication and signing of multicast streams over lossy channels,” in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pp. 56–73, IEEE, 2000.

[34] C. Szilagy and P. Koopman, “A flexible approach to embedded network multicast authentication,” 2008.

[35] M. Wolf, A. Weimerskirch, and T. Wollinger, “State of the art: Embedding security in vehicles,” *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, p. 074706, 2007.

[36] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Moderator-Ravi, “Security as a new dimension in embedded system design,” in *Proceedings of the 41st annual Design Automation Conference*, pp. 753–760, ACM, 2004.

[37] P. Kleberger, T. Olovsson, and E. Jonsson, “Security aspects of the in-vehicle network in the connected car,” in *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pp. 528–533, IEEE, 2011.

[38] K. Han, A. Weimerskirch, and K. G. Shin, “Automotive cybersecurity for in-vehicle communication,” in *IQT QUARTERLY*, vol. 6, pp. 22–25, 2014.

[39] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2, pp. 119–129, IEEE, 2000.

[40] "Buffer overflow exploit - dhaval kapil." https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/. (Accessed on 08/06/2017).

[41] J. Habibi, A. Panicker, A. Gupta, and E. Bertino, "Disarm: mitigating buffer overflow attacks on embedded devices," in *International Conference on Network and System Security*, pp. 112–129, Springer, 2015.

[42] C. Cowan, "Buffer overflow attacks," *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*, 2011.

[43] D. Evans, a. Nguyen-Tuong, and J. Knight, "Effectiveness of moving target defenses," *Moving Target Defense: An Asymmetric Approach to Cyber Security*, 2011.

[44] H. Okhravi, M. A. Rabe, T. J. Mayberry, W. G. Leonard, T. R. Hobson, D. Bigelow, and W. W. Streilein, "Survey of Cyber Moving Targets," *Lincoln Laboratory Technical Report*, 2013.

[45] H. Alnabulsi, Q. Mamun, R. Islam, and M. U. Chowdhury, "Defence against code injection attacks," in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 2015.

[46] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

[47] S. W. Boyd, G. S. Kc, M. E. Locasto, A. D. Keromytis, and V. Prevelakis, "On the general applicability of instruction-set randomization," *IEEE Transactions on Dependable and Secure Computing*, 2010.

[48] Y. Weiss and E. G. Barrantes, "Known/Chosen key attacks against software Instruction Set Randomization," in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2006.

[49] A. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? the effectiveness of instruction set randomization," *14th USENIX Security Symposium*, 2005.

[50] K. Sinha, V. Kemerlis, V. Pappas, S. Sethumadhavan, and A. D. Keromytis, "Enhancing security by diversifying instruction sets," 2014.

[51] 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 - CCS '13*, 2013.

[52] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," *Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[53] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[54] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, pp. 190–200, ACM, 2005.

[55] K. Scott and J. Davidson, "Strata: A software dynamic translation infrastructure," in *IEEE Workshop on Binary Translation*, 2001.

[56] C. Gorgovan, A. D'antras, and M. Luján, "Mambo: a low-overhead dynamic binary modification tool for arm," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 14, 2016.

[57] L. Li, J. E. Just, and R. Sekar, "Address-space randomization for windows systems," in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2006.

[58] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 574–588, IEEE, 2013.

[59] Z. Wang, R. Cheng, and D. Gao, "Revisiting address space randomization," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011.

[60] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits.," in *USENIX Security Symposium*, vol. 12, pp. 291–301, 2003.

[61] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address space randomization for mobile devices," *Proceedings of the fourth ACM conference on Wireless network security - WiSec '11*, 2011.

[62] 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 - CCS '04*, 2004.

[63] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 2016.

[64] S. Bhatkar and R. Sekar, "Data space randomization," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.

[65] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," tech. rep., Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, 2008.

[66] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar, "Eternal war in memory," *IEEE Security & Privacy*, vol. 12, no. 3, pp. 45–53, 2014.

[67] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12, pp. 91–104, 2003.

[68] M. Larsen and F. Gont, "Recommendations for Transport-Protocol Port Randomization," *RFC 6051*, 2011.

[69] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Computer Networks*, 2007.

[70] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal, "Investigating the application of moving target defenses to network security," in *Proceedings - 2013 6th International Symposium on Resilient Control Systems, ISRCS 2013*, 2013.

[71] A. R. Chavez, W. M. Stout, and S. Peisert, "Techniques for the dynamic randomization of network attributes," in *Proceedings - International Carnahan Conference on Security Technology*, 2016.

[72] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. September, pp. 1–58, 2009.

[73] M. M. L. Prasanthi, "Cyber Crime: Prevention & Detection," *Ijarcce*, vol. 4, no. 3, pp. 45–48, 2015.

[74] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[75] B. Sprunt, L. Sha, and J. Lehoczky, "Scheduling sporadic and aperiodic events in a hard real-time system," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1989.

[76] F. Jahanian, "Fault-tolerance in embedded real-time systems," in *Hardware and Software Architectures for Fault Tolerance*, pp. 237–249, Springer, 1994.

[77] D. K. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *IEEE Transactions on computers*, vol. 43, no. 10, pp. 1163–1174, 1994.

[78] J. Xu and B. Randell, "Roll-forward error recovery in embedded real-time systems," in *Parallel and Distributed Systems, 1996. Proceedings., 1996 International Conference on*, pp. 414–421, IEEE, 1996.

[79] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, pp. 2–11, 1985.

[80] P. A. Lee and T. Anderson, *Fault tolerance: principles and practice*, vol. 3. Springer Science & Business Media, 2012.

[81] J. Specht, "Redundancy, fault tolerance, terminology." http://www.ieee802.org/1/files/public/docs2013/new-tsn-specht-redundancy-terminology-20130115-v01.pdf. (Accessed on 11/08/2017).

[82] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254, IEEE Computer Society, 2005.

[83] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentell, "Fault tolerance techniques for wireless ad hoc sensor networks," in *Sensors, 2002. Proceedings of IEEE*, vol. 2, pp. 1491–1496, IEEE, 2002.

[84] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2006.

[85] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[86] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *Reliable Software TechnologiesAda-Europe'96*, pp. 38–57, Springer, 1996.

[87] A. Fedoruk and R. Deters, "Improving fault-tolerance by replicating agents," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pp. 737–744, ACM, 2002.

[88] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[89] A. Avizienis and J. P. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, no. 8, pp. 67–80, 1984.

[90] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Hardware-and software-fault tolerance," in *ESPRIT90*, pp. 786–789, Springer, 1990.

[91] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[92] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[93] E. Ayanoglu, I. Chih-Lin, R. D. Gitlin, and J. E. Mazo, "Diversity coding for transparent self-healing and fault-tolerant communication networks," *IEEE Transactions on communications*, vol. 41, no. 11, pp. 1677–1686, 1993.

[94] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with sql database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, 2007.

[95] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[96] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on software Engineering*, no. 1, pp. 23–31, 1987.

[97] B. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach," in *Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on*, pp. 3–12, IEEE, 1988.

[98] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[99] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[100] J. Rushby, "Reconfiguration and transient recovery in state machine architectures," in *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pp. 6–15, IEEE, 1996.

[101] C.-S. Li and R. Ramaswami, "Automatic fault detection, isolation, and recovery in transparent all-optical networks," *Journal of Lightwave Technology*, vol. 15, no. 10, pp. 1784–1793, 1997.

[102] D. Nguyen and D.-B. Liu, "Recovery blocks in real-time distributed systems," in *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*, pp. 149–154, IEEE, 1998.

[103] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Operating Systems*, pp. 171–187, Springer, 1974.

[104] T. Anderson and R. Kerr, "Recovery blocks in action: A system supporting high reliability," in *Proceedings of the 2nd international conference on Software engineering*, pp. 447–457, IEEE Computer Society Press, 1976.

[105] S. K. Shrivastava and A. A. Akinpelu, "Fault-tolerant sequential programming using recovery blocks," in *Reliable Computer Systems*, pp. 112–114, Springer, 1985.

[106] K. G. Shin and Y.-H. Lee, "Evaluation of error recovery blocks used for cooperating processes," *IEEE Transactions on Software Engineering*, no. 6, pp. 692–700, 1984.

[107] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pp. 65–74, ACM, 2013.

[108] L. Sha, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[109] J. Yao, X. Liu, G. Zhu, and L. Sha, "Netsimplex: Controller fault tolerance architecture in networked control systems," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 346–356, 2013.

[110] X. Wang, N. Hovakimyan, and L. Sha, "L1simplex: fault-tolerant control of cyber-physical systems," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, pp. 41–50, ACM, 2013.

[111] D. Seto, E. Ferreira, and T. F. Marz, "Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis)," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.

[112] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, “The system-level simplex architecture for improved real-time embedded system safety,” in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pp. 99–107, IEEE, 2009.

[113] M.-K. Yoon, B. Liu, N. Hovakimyan, and L. Sha, “Virtualdrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems,” in *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pp. 143–154, ACM, 2017.

[114] O. Scheickl and M. Rudorfer, “Automotive real time development using a timing-augmented autosar specification,” *Proceedings of ERTS2008*, vol. 4, 2008.

[115] B. Bonakdarpour, “Challenges in transformation of existing real-time embedded systems to cyber-physical systems,” *ACM SIGBED Review*, vol. 5, no. 1, p. 11, 2008.

[116] C.-W. Lin, Q. Zhu, C. Phung, and A. Sangiovanni-Vincentelli, “Security-aware mapping for can-based real-time distributed automotive systems,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 115–121, IEEE Press, 2013.

[117] C.-W. Lin, Q. Zhu, and A. Sangiovanni-Vincentelli, “Security-aware modeling and efficient mapping for can-based real-time distributed automotive systems,” *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 11–14, 2015.

[118] S. Woo, H. J. Jo, and D. H. Lee, “A practical wireless attack on the connected car and security protocol for in-vehicle can,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 993–1006, 2015.

[119] M. Lukasiewycz, P. Mundhenk, and S. Steinhorst, “Security-aware obfuscated priority assignment for automotive can platforms,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 2, p. 32, 2016.

[120] C. Miller, “Car hacking: You cannot have safety without security.” http://resources.infosecinstitute.com/car-hacking-safety-without-security/#gref. (Accessed on 01/26/2017).

[121] T. Hoppe, S. Kiltz, and J. Dittmann, “Security threats to automotive can networks–practical examples and selected short-term countermeasures,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 235–248, Springer, 2008.

[122] C. Lu, “Security of autonomous vehciles.” http://www.cse.wustl.edu/~jain/cse571-14/ftp/vehicle_security.pdf. (Accessed on 01/26/2017).

[123] M. Xia, P. J. Antsaklis, and V. Gupta, “Passivity indices and passivation of systems with application to systems with input/output delay,” in *53rd IEEE Conference on Decision and Control*, pp. 783–788, 2014.

[124] X. Koutsoukos, N. Kottenstette, J. Hall, E. Eyisi, H. Leblanc, J. Porter, and J. Sztipanovits, “A passivity approach for model-based compositional design of networked control systems,” *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 4, p. 75, 2012.

[125] N. Kottenstette, J. F. Hall, X. Koutsoukos, J. Sztipanovits, and P. Antsaklis, “Design of networked control systems using passivity,” *IEEE Transactions on Control Systems Technology*, vol. 21, no. 3, pp. 649–665, 2013.

[126] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha, “Real-time reachability for verified simplex design,” in *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pp. 138–148, IEEE, 2014.

[127] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, “Software monitoring with controllable overhead,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 327–347, 2012.

[128] A. Bauer, M. Leucker, and C. Schallhart, “Model-based runtime analysis of distributed reactive systems,” in *Software Engineering Conference, 2006. Australian*, pp. 10–pp, IEEE, 2006.

[129] S. Tripakis, “A combined on-line/off-line framework for black-box fault diagnosis,” in *International Workshop on Runtime Verification*, pp. 152–167, Springer, 2009.

[130] M. Wagner, P. Koopman, J. Bares, and C. Ostrowski, “Building safer ugvs with run-time safety invariants,” in *National Defense Industrial Association Systems Engineering Conference*, 2009.

[131] F. B. Schneider, “Enforceable security policies,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[132] J. Ligatti, L. Bauer, and D. Walker, “Run-time enforcement of nonsafety policies,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 3, p. 19, 2009.

[133] Y. Falcone, J.-C. Fernandez, and L. Mounier, “What can you verify and enforce at runtime?,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 349–382, 2012.

[134] M. Clark, X. Koutsoukos, J. Porter, R. Kumar, G. Pappas, O. Sokolsky, I. Lee, and L. Pike, “A study on run time assurance for complex cyber physical systems,” tech. rep., AIR FORCE RESEARCH LAB WRIGHT-PATTERSON AFB OH AEROSPACE SYSTEMS DIR, 2013.

[135] D. E. Simon, *An embedded software primer*, vol. 1. Addison-Wesley Professional, 1999.

[136] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, “The worst-case execution-time problemoverview of methods and survey of tools,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[137] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.

[138] R. W. Butler, “A primer on architectural level fault tolerance,” 2008.

[139] F. Jahanian, R. Rajkumar, and S. C. Raju, “Runtime monitoring of timing constraints in distributed real-time systems,” *Real-Time Systems*, vol. 7, no. 3, pp. 247–273, 1994.

[140] R. Alur and T. A. Henzinger, “Real-time logics: Complexity and expressiveness,” *Information and Computation*, vol. 104, no. 1, pp. 35–77, 1993.

[141] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[142] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, “Efficient techniques for near-optimal instrumentation in time-triggered runtime verification,” in *International Conference on Runtime Verification*, pp. 208–222, Springer, 2011.

[143] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, “Hardware runtime monitoring for dependable cots-based real-time embedded systems,” in *Real-Time Systems Symposium, 2008*, pp. 481–491, IEEE, 2008.

[144] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok, “Runtime verification with state estimation,” in *International Conference on Runtime Verification*, pp. 193–207, Springer, 2011.

[145] R. Obermaisser, *Event-triggered and time-triggered control paradigms*, vol. 22. Springer Science & Business Media, 2004.

[146] H. Kopetz, “Why time-triggered architectures will succeed in large hard real-time systems,” in *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pp. 2–9, IEEE, 1995.

[147] J. Rushby, “Bus architectures for safety-critical embedded systems,” in *International Workshop on Embedded Software*, pp. 306–323, Springer, 2001.

[148] D. Isovic and G. Fohler, “Handling sporadic tasks in off-line scheduled distributed real-time systems,” in *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS’99*, pp. 60–67, IEEE, 1999.

[149] A. Zuepke, M. Bommert, and D. Lohmann, “Autobest: a united autosar-os and arinc 653 kernel,” in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 133–144, IEEE, 2015.

[150] P. J. Prisaznuk, “Arinc 653 role in integrated modular avionics (ima),” in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pp. 1–E, IEEE, 2008.

[151] N. Diniz and J. Rufino, “Arinc 653 in space,” in *DASIA 2005-Data Systems in Aerospace*, vol. 602, 2005.

[152] A. Dubey, G. Karsai, and N. Mahadevan, “A component model for hard real-time systems: Ccm with arinc-653,” *Software: Practice and Experience*, vol. 41, no. 12, pp. 1517–1550, 2011.

[153] A. One, “Smashing the stack for fun and profit (1996),” *See http://www. phrack. org/show. php*, 2007.

[154] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 745–762, IEEE, 2015.

[155] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[156] G. Portokalidis and A. D. Keromytis, “Fast and practical instruction-set randomization for commodity systems,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 41–48, ACM, 2010.

[157] H. Marco-Gisbert and I. Ripoll, “On the effectiveness of full-aslr on 64-bit linux,” 2014.

[158] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, “Secure and practical defense against code-injection attacks using software dynamic translation,” in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 2–12, ACM, 2006.

[159] “securing_self_driving_cars.pdf.” http://illmatics.com/securing_self_driving_cars.pdf. (Accessed on 12/05/2018).

[160] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 48–62, IEEE, 2013.

[161] “Jetson tk1 - elinux.org.” http://elinux.org/Jetson_TK1. (Accessed on 06/03/2017).

[162] G. Coley, “Beaglebone black system reference manual,” *Texas Instruments, Dallas*, 2013.

[163] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, “Torcs, the open racing car simulator,” *Software available at http://torcs. sourceforge. net*, vol. 4, 2000.

[164] K. Tindell, A. Burns, and A. J. Wellings, “Calculating controller area network (can) message response times,” *Control engineering practice*, vol. 3, no. 8, pp. 1163–1169, 1995.

[165] “End-to-end deep learning for self-driving cars.” https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/. (Accessed on 06/04/2017).

[166] "Github - udacity/self-driving-car-sim: A self-driving car simulator built with unity." https://github.com/udacity/self-driving-car-sim. (Accessed on 06/03/2017).

[167] V. Hilderman and T. Baghi, *Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware)*. Avionics Communications, 2007.

[168] P. Anderson, "Coding standards for high-confidence embedded systems," in *MILCOM 2008-2008 IEEE Military Communications Conference*, pp. 1–7, IEEE, 2008.

[169] C. Lattner *et al.*, "The llvm compiler infrastructure," *URL http://llvm. org*, 2010.

[170] A. Dinaburg and A. Ruef, "Mcsema: Static translation of x86 instructions to llvm," in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[171] F. Markl, "Case study on llvm as suitable intermediate language for binary analysis," *ret*, vol. 32, p. 0.

[172] M. CAPELLETTI, "Unlinker: an approach to identify original compilation units in stripped binaries," 2017.

[173] I. P. Disassembler, "Debugger," 2010.

[174] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[175] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[176] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, pp. 265–266, ACM, 2016.

[177] "Overview llvm 10 documentation." https://llvm.org/docs/. (Accessed on 08/05/2019).

[178] S. Bhatkar and R. Sekar, "Data space randomization," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 1–22, Springer, 2008.

[179] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," *arXiv preprint arXiv:1711.03938*, 2017.

[180] "adubey14/arinc653emulator: This code base contains a linux emulator for the arinc-653 operating system services." https://github.com/adubey14/arinc653emulator. (Accessed on 07/07/2019).