Microprocessors and Microsystems 73 (2020) 102954

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Integrated moving target defense and control reconfiguration for securing Cyber-Physical systems

Bradley Potteiger[a,*], Zhenkai Zhang[b], Xenofon Koutsoukos[a]

[a] Vanderbilt University Nashville, TN United States
[b] Texas Tech University Lubbock, TX United States

ARTICLE INFO

Article history:
Received 29 April 2019
Revised 21 October 2019
Accepted 11 December 2019
Available online 17 December 2019

Keywords:
Moving target defenses
Cyber-Physical systems
Resiliency
Instruction set randomization
Address space randomization

ABSTRACT

With the increasingly connected nature of Cyber-Physical Systems (CPS), new attack vectors are emerging that were previously not considered in the design process. Specifically, autonomous vehicles are one of the most at risk CPS applications, including challenges such as a large amount of legacy software, non-trusted third party applications, and remote communication interfaces. With zero day vulnerabilities constantly being discovered, an attacker can exploit such vulnerabilities to inject malicious code or even leverage existing legitimate code to take over the cyber part of a CPS. Due to the tightly coupled nature of CPS, this can lead to altering physical behavior in an undesirable or devastating manner. Therefore, it is no longer effective to reactively harden systems, but a more proactive approach must be taken. Moving target defense (MTD) techniques such as instruction set randomization (ISR), and address space randomization (ASR) have been shown to be effective against code injection and code reuse attacks. However, these MTD techniques can result in control system crashing which is unacceptable in CPS applications since such crashing may cause catastrophic consequences. Therefore, it is crucial for MTD techniques to be complemented by control reconfiguration to maintain system availability in the event of a cyber-attack. This paper addresses the problem of maintaining system and security properties of a CPS under attack by integrating moving target defense techniques, as well as detection, and recovery mechanisms to ensure safe, reliable, and predictable system operation. Specifically, we consider the problem of detecting code injection as well as code reuse attacks, and reconfiguring fast enough to ensure the safety and stability of autonomous vehicle controllers are maintained. By using MTD such as ISR, and ASR, our approach provides the advantage of preventing attackers from obtaining the reconnaissance knowledge necessary to perform code injection and code reuse attacks, making sure attackers can't find vulnerabilities in the first place. Our system implementation includes a combination of runtime MTD utilizing AES 256 ISR and fine grained ASR, as well as control management that utilizes attack detection, and reconfiguration capabilities. We evaluate the developed security architecture in an autonomous vehicle case study, utilizing a custom developed hardware-in-the-loop testbed.

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 [42]. 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 [8].

* Corresponding author.
E-mail addresses: bradley.d.potteiger@vanderbilt.edu (B. Potteiger), zhenkai.zhang@ttu.edu (Z. Zhang), xenofon.koutsoukos@vanderbilt.edu (X. Koutsoukos).

https://doi.org/10.1016/j.micpro.2019.102954
0141-9331/

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 [27]. 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 [35]. One of the most popular examples of this type of attack is return oriented programming (ROP) [34] 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 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 Defenses (MTD) aim 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 [28]. 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 [24], and fine grained [41], while also having the ability to be customized to protect the most critical memory segments [47].

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 [14], combined with a fine grained ASR implementation for protecting against the relative, and direct control flow redirection necessary for code reuse attacks [30]. Our security architecture consists of three stages including attack protection (randomize, derandomize), detection, and recovery. The main CPS challenge addressed in this paper is protecting system integrity during cyber-attacks, while maintaining system availability with safe and reliable operation. Our paper 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.

The paper is organized as follows. Section 2 introduces the system and attack model utilized throughout the paper. Section 3 describes the high level component organization of our security architecture. Section 4 describes the implementation of our security framework including the MTD implementation, and process flow during a cyber-attack. Section 5 describes the evaluation of our security architecture including a developed hardware-in-the-loop testbed, and autonomous vehicle case study. Section 6 describes current limitations of our security framework, as well as our plans to address them. Section 7 describes related work for our paper. Finally, Section 8 ends the paper with concluding remarks.

2. System model

An exemplary vehicle system model is shown in Fig. 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.

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 [2] 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.

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 [25].

In the rest of this paper 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.

3. Architecture

In Fig. 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

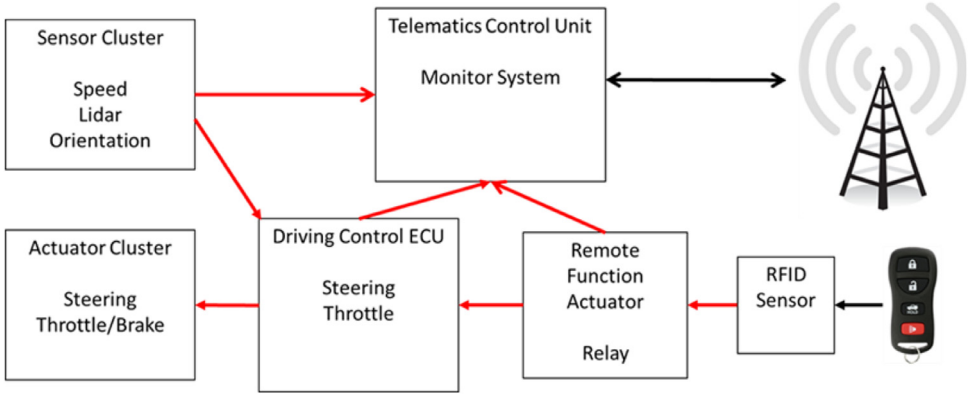

Fig. 1. Vehicle Architecture Diagram.

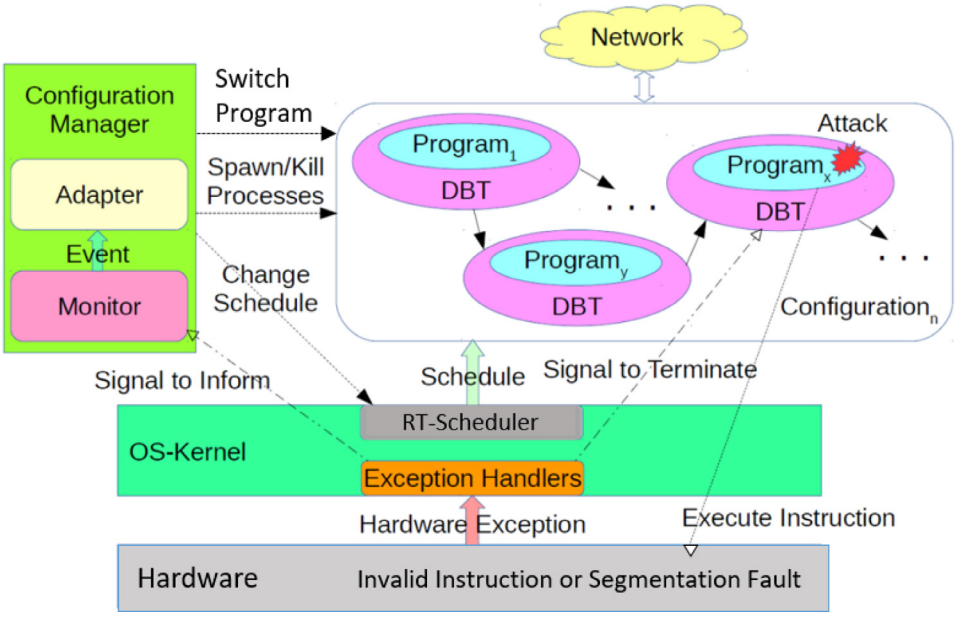

Fig. 2. Control Architecture [30].

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.

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.

4.1. MTD Implementation

4.1.1. *ISR*

To perform a successful code injection attack, an adversary must have knowledge of the system instruction set architecture to craft a valid payload [14]. 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.

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.

Algorithm 1 PAG Integrity Check.

```
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
```

4.1.2. *ASR*

To perform a successful code reuse attack, an adversary must have knowledge of the memory layout, specifically the locations of safety-critical functions [43]. 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 [6]). 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.

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.

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
```

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 [11].

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 execution. 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 CPS controller. The second assumption is that the communication between the Configuration Manager and the DBT processes must

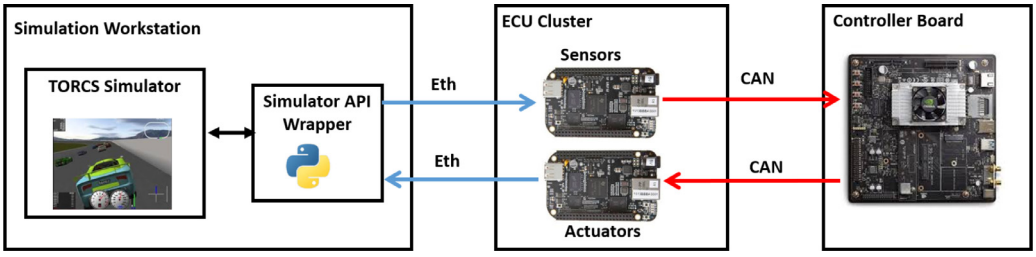

Fig. 3. Testbed Hardware Architecture.

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.

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.

5. Evaluation

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.

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 [10] 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 [9] 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 Fig. 3.

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 [49]. 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.

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 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 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. [44] 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 Fig. 4.

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.

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.

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

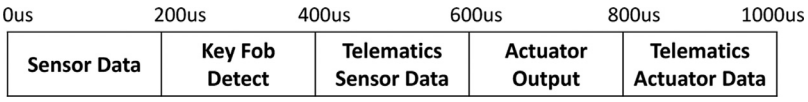

Fig. 4. CAN Bus Message Timeslots.

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.

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.

5.4. Overhead results

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 Figs. 5, and 6, the overhead created with both ISR and ASR enabled is minimal enough to maintain execution

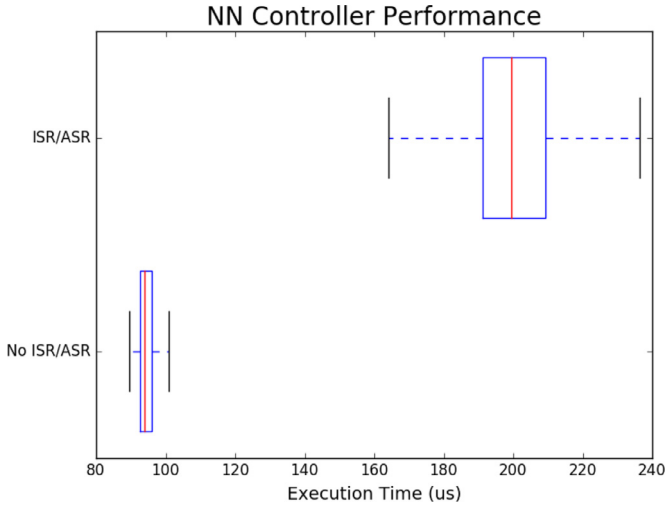

Fig. 5. Neural Network Controller Execution Times.

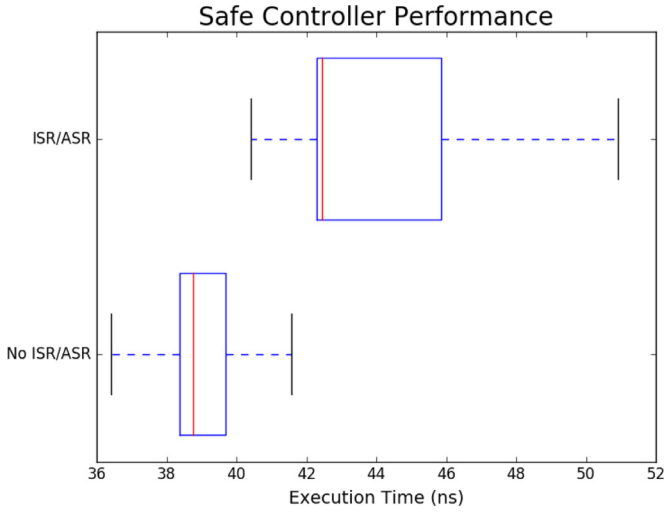

Fig. 6. Safe Controller Execution Times.

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.

5.5. Worst case recovery time

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. Fig. 7 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 50 $ms + 52$ ns after the last actuation command, essentually equating to just after the next

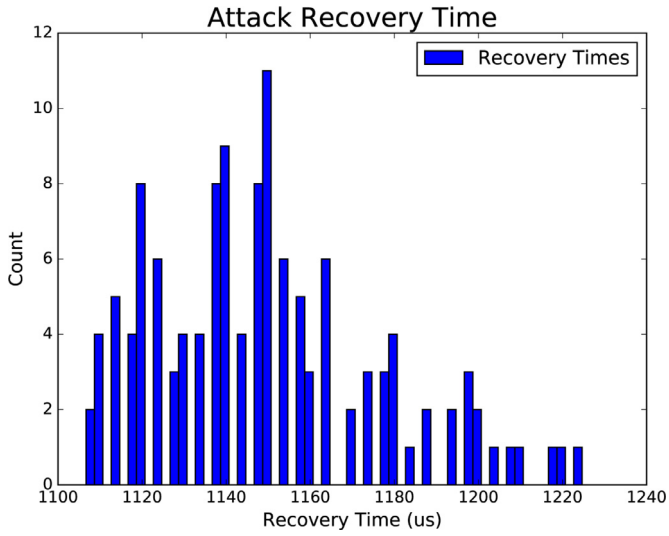

Fig. 7. Attack Recovery Times.

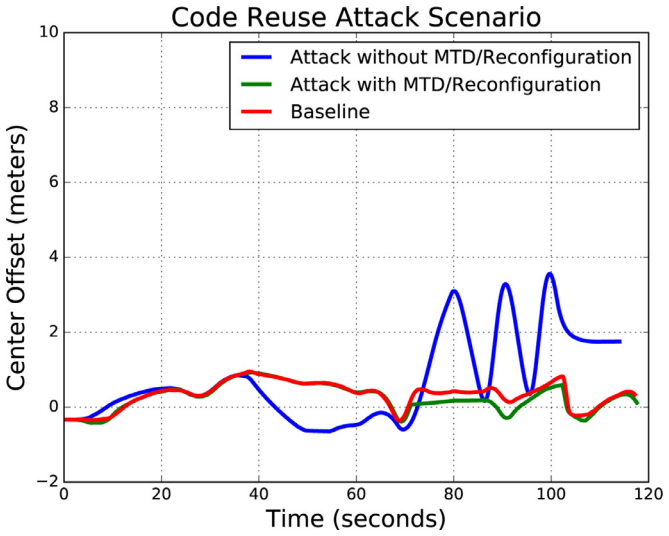

Fig. 9. Code Reuse Scenario Road Center Offset Time Plot.

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.

5.6. Safety-Critical results

Fig. 8 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.

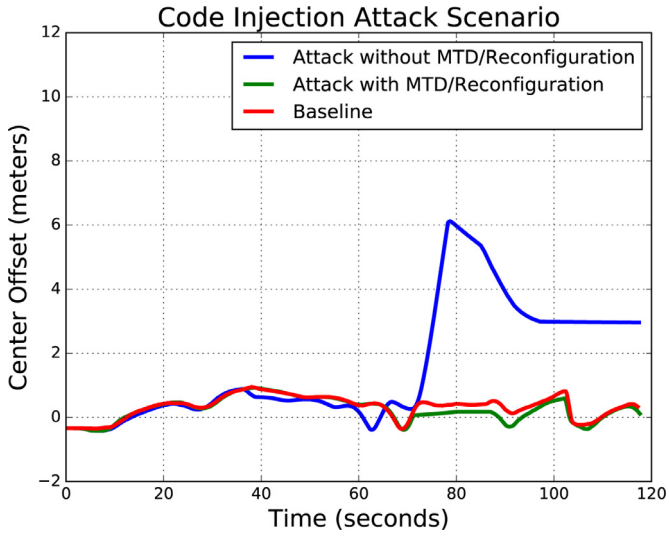

Fig. 8. Vehicle Road Center Offset Time Plot.

In the second scenario where a code reuse attack is executed, Fig. 9 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.

6. Limitations

Under the current implementation of our security architecture, there are a few limitations. These limitations are described below, as well as our plans to address them in the future.

During the normal reconfiguration process, once an attack is detected, an execution transferring process takes place to restore execution to the backup CPS controller. At this point a new default controller instance is spawned to serve as the new backup controller in the architecture. However, this process requires a minimum amount of recovery time to ensure that both controllers are fully loaded and operational. In the case of a rapid attack campaign, partial protection is provided by leveraging different CPS controllers with different software structure. This means that attackers can't infiltrate both controllers with the same attack techniques. However, if a new vulnerability is found in the backup controller, another recovery process can ensue. If a rapid attack campaign occurs, the system could be forced to remain in a constant recovery state, potentially leading to denial of service behavior. If this behavior occurs, a fail safe mechanism can be implemented to stop the vehicle at a safe position on the road, and wait for further assistance from the operators.

For our current implementation, a DBT is needed for customizing the runtime environment of the CPS controller. Even though several benefits have been demonstrated with the introduction of

fine grained ASR compared to course grained ASR, the utilization of a DBT provides a degree of performance overhead that could be a limiting factor in applications with tight real time deadlines. As such, a design time decision is needed to determine the applicability of utilizing this type of approach with respect to safely handling this performance overhead.

For this work, we focus on two popular attack vectors including code injection, and code reuse attacks. The MTD techniques of ISR, and ASR are effective in mitigating against these types of attacks. However, another popular attack vector is a non-control data attack. These types of attack can overwrite adjacent safety-critical variables for the purpose of altering the computation output of controllers. To address this attack vector, we plan to implement data space randomization and integrity checking mechanisms to ensure the authenticity of safety-critical variables in our program [31].

To ensure the integrity of our security architecture, the randomization keys must remain secure. In some cases, side channel attacks have been shown to be effective techniques to reverse engineer software to potentially defeat the randomization of CPS controllers [18,21]. Currently, we randomize CPS controllers with different keys to provide a basic protection against a widespread side channel attack effect. However, in the future we plan on implementing dynamic reconfiguration to periodically update the randomization keys of controllers to mitigate against reconnaissance efforts through side channel means.

7. Related work

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 [19]. Numerous vulnerabilities have been found in current automobile models [1,25]. Automobile security research generally focuses on hardening systems based on finding vulnerabilities through penetration testing, and implementing defense solutions such as securing internal network communication [20], implementing encryption, and authentication of data [17], and securing external interfaces [13]. However, these methods tend to be vulnerable to zero day exploits which leverage vulnerabilities not known at design time. Once an adversary can gain entry to the internal vehicle network through vulnerabilities in external device interfaces such as the infotainment center, they have the ability to interact with several safety-critical ECUs throughout the system.

Memory corruption can be broken into four categories: code corruption, control flow hijacking, information leakage and non-control data attacks [43]. Exploitation often involves leveraging programming bugs such as dangling pointers, integer, and buffer overflows. Defense techniques for code injection attacks have been proposed including $W \oplus X$ protections such as DEP [48]. However, in response to these defense techniques, there has been a rise in ROP based attacks, leveraging existing code segments to accomplish attacker goals without the need for injecting customized instruction payloads [34]. Memory corruption attacks today have been found to still be a significant threat, despite decades of research in defense protections, and the development of safe programming languages [45]. In applications such as automobiles where C/C++ legacy code still makes up the majority of software, these types of vulnerabilites will exist for years to come.

ISR implementations range from hardware based FPGA implementations [40], to software implementaitons based on virtualization in the operating system. Past implementations of ISR have been based on utilizing emulators for customizing processor architecture representations [5,16]. However, ISR software implementations are based on DBT tools such as MAMBO [11], STRATA [36], and PIN [22] that use dynamic instrumentation to create a virtualization environment that can modify a program dynamically at runtime, including the ability to alter instructions as they are fetched. As such, programs can be randomized dynamically at load time, and derandomized instruction by instruction as they are fetched by utilizing a generated randomization key [29]. One recent ISR implementation has been developed for Intel X86 machines utilizing the PIN DBT [29], while another implementation focuses on the combination of ISR and course grained ASR with respect to randomizing system calls [15]. However, our framework is the first ISR implementation on ARM based systems utilizing the DBT approach.

ASR has been implemented on Linux [6], Windows [19], Macintosh [7], and mobile platforms [7]. The Linux version of ASR, referred to as address space layout randomization (ASLR), is the most widely utilized implementation, randomizing the base addresses of shared libraries, the stack, and heap by default. However, it is noted that on 32 bit Linux systems there are only 16 bits of randomization and on 64 bit Linux systems there are 32 bits of randomization [39]. There have been a couple of fine grained ASR research prototypes including MARLIN [12]. However, our implementation is the first to include both ISR, and fine grained ASR on ARM based systems. Our view is that this will allow our framework to be applicable to the CPS, as the majority of these devices tend to include ARM processors. [43] provides a good analysis of the benefits of MTD techniques such as ISR, and ASR against memory corruption attacks like code corruption, and control flow hijacking. ASR was found to be the most prominent probabilistic MTD technique against control flow hijacking attacks. There is a trade-off however in the average 10% performance overhead required for position independent compilation needed for the implementation, compared to the increase in security. The importance of reducing information leakage in a program was also emphasized for minimizing the probability that the randomization key will be reverse engineered.

With regards to recovery, there has been a wealth of work in the area of software fault tolerance. Several existing methodologies integrate N-Version programming to lower the probability of successive attacks by implementing different software versions with different structures, but similar semantics [3]. Additionally, checkpointing techniques such as recovery blocks have been utilized for rollback recovery implementations, allowing for controllers to maintain state through the reconfiguration process [23,32,33]. Simplex, which is the primary motivator of our security architecture, has been a widely utilized fault tolerant architecture, which consists of a complex controller, safety controller, and decision module which switches execution between the two based on specific events [38]. Several previous simplex based implementations include Secure System Simplex [26], Net Simplex [50], and L1 Simplex [46]. Furthermore, simplex architectures have been popular in safety-critical applications such as flight control systems [37], medical devices [4], and unmanned aerial vehicles [51].

8. Conclusion

In this work, 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 main-

tain 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 future, we plan on integrating data space randomization, dynamic reconfiguration, and time triggered functionality.

Declaration of competing interest

The authors certify that they have NO affiliations with or involvement in any organization or entity with any financila interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

Acknowledgements

This work is supported in part by the Air Force Research Laboratory (FA 8750-14-2-0180), the National Science Foundation (CNS-1739328, CNS-1238959), and by NIST (70NANB17H266). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL, NSF, or NIST.

References

[1] New vehicle security research by keenlab: experimental security assessment of bmw cars - keen security lab blog, (https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/).
[2] securing_self_driving_cars.pdf, (http://illmatics.com/securing_self_driving_cars.pdf) (Accessed on 12/05/2018).
[3] A. Avizienis, The n-version approach to fault-tolerant software, IEEE Trans. Softw. Eng. (12) (1985) 1491–1501.
[4] S. Bak, D.K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, 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, IEEE, 2009, pp. 99–107.
[5] E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, D.D. Zovi, Randomized instruction set emulation to disrupt binary code injection attacks, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, ACM, 2003, pp. 281–289.
[6] S. Bhatkar, D.C. DuVarney, R. Sekar, Address obfuscation: an efficient approach to combat a broad range of memory error exploits., in: USENIX Security Symposium, 12, 2003, pp. 291–301.
[7] H. Bojinov, D. Boneh, R. Cannings, I. Malchev, Address space randomization for mobile devices, Proc. Fourth ACM Conf. Wirel. Netw. Secur. - WiSec '11 (2011), doi:10.1145/1998412.1998434.
[8] 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., USENIX Security Symposium, San Francisco, 2011.
[9] G. Coley, Beaglebone black system reference manual, Texas Instrument. Dallas (2013).
[10] D. Franklin, Nvidia jetson tx2 delivers twice the intelligence to the edge, NVIDIA Accelerat. Comput. – Parallel Forall (2017).
[11] C. Gorgovan, A. D'antras, M. Luján, Mambo: a low-overhead dynamic binary modification tool for arm, ACM Trans. Archit. Code Optim. (TACO) 13 (1) (2016) 14.
[12] A. Gupta, S. Kerr, M.S. Kirkpatrick, E. Bertino, Marlin: a fine grained randomization approach to defend against rop attacks, in: International Conference on Network and System Security, Springer, 2013, pp. 293–306.
[13] K. Han, A. Weimerskirch, K.G. Shin, Automotive cybersecurity for in-vehicle communication, in: IQT QUARTERLY, 6, 2014, pp. 22–25.
[14] W. Hu, J. Hiser, D. Williams, A. Filipi, J.W. Davidson, D. Evans, J.C. Knight, A. Nguyen-Tuong, J. Rowanhill, Secure and practical defense against code-injection attacks using software dynamic translation, in: Proceedings of the 2nd International Conference on Virtual Execution Evironments, ACM, 2006, pp. 2–12.
[15] X. Jiang, H.J. Wangz, D. Xu, Y.-M. Wang, Randsys: Thwarting code injection attacks with system service interface randomization, in: Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on, IEEE, 2007, pp. 209–218.
[16] G.S. Kc, A.D. Keromytis, V. Prevelakis, Countering code-injection attacks with instruction-set randomization, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, ACM, 2003, pp. 272–280.
[17] P. Kleberger, T. Olovsson, E. Jonsson, Security aspects of the in-vehicle network in the connected car, in: Intelligent Vehicles Symposium (IV), 2011 IEEE, IEEE, 2011, pp. 528–533.
[18] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: exploiting speculative execution, arXiv:1801.01203(2018).
[19] L. Li, J.E. Just, R. Sekar, Address-space randomization for windows systems, in: Proceedings - Annual Computer Security Applications Conference, ACSAC, 2006, doi:10.1109/ACSAC.2006.10.
[20] C.-W. Lin, A. Sangiovanni-Vincentelli, Cyber-security for the controller area network (can) communication protocol, in: Cyber Security (CyberSecurity), 2012 International Conference on, IEEE, 2012, pp. 1–7.
[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown, ArXiv e-prints (2018).
[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Acm Sigplan Notices, 40, ACM, 2005, pp. 190–200.
[23] M.R. Lyu, Software Fault Tolerance, John Wiley & Sons, Inc., 1995.
[24] H. Marco-Gisbert, I. Ripoll, On the effectiveness of full-aslr on 64-bit linux, In-depth security conference, DeepSec, 2014.
[25] C. Miller, C. Valasek, Remote exploitation of an unaltered passenger vehicle, Black Hat USA, 2015, 2015.
[26] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, 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, ACM, 2013, pp. 65–74.
[27] A. One, Smashing the stack for fun and profit (1996), See http://www.phrack.org/show.php (2007).
[28] G. Portokalidis, A.D. Keromytis, Fast and practical instruction-set randomization for commodity systems, in: Proceedings of the 26th Annual Computer Security Applications Conference, ACM, 2010, pp. 41–48.
[29] G. Portokalidis, A.D. Keromytis, Fast and practical instruction-set randomization for commodity systems, in: Proceedings of the 26th Annual Computer Security Applications Conference, ACM, 2010, pp. 41–48.
[30] B. Potteiger, Z. Zhang, 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, ACM, 2018, p. 5.
[31] B. Potteiger, Z. Zhang, X. Koutsoukos, Integrated data space randomization and control reconfiguration for securing cyber-physical systems, in: Proceedings of the 6th Annual Symposium and Bootcamp on Hot Topics in the Science of Security, ACM, 2019, p. 5.
[32] L.L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, 2001.
[33] B. Randell, System structure for software fault tolerance, IEEE Trans. Soft. Eng. (2) (1975) 220–232.
[34] R. Roemer, E. Buchanan, H. Shacham, S. Savage, Return-oriented programming: systems, languages, and applications, ACM Trans. Inf. Syst. Secur. (TISSEC) 15 (1) (2012) 2.
[35] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, 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, IEEE, 2015, pp. 745–762.
[36] K. Scott, J. Davidson, Strata: a software dynamic translation infrastructure, IEEE Workshop on Binary Translation, 2001.
[37] D. Seto, E. Ferreira, T.F. Marz, Case study: development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis), Technical Report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.
[38] L. Sha, Using simplicity to control complexity, IEEE Softw. 18 (4) (2001) 20–28.
[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, in: Proceedings of the 11th ACM Conference on Computer and Communications Security - CCS '04, 2004, doi:10.1145/1030083.1030124.

[40] K. Sinha, V. Kemerlis, V. Pappas, S. Sethumadhavan, A.D. Keromytis, Enhancing security by diversifying instruction sets, Coumbia University Report (2014).
[41] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, 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, IEEE, 2013, pp. 574–588.
[42] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaâniche, 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, IEEE, 2013, pp. 1–12.
[43] L. Szekeres, M. Payer, T. Wei, D. Song, Sok: eernal war in memory, in: Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 48–62.
[44] K. Tindell, A. Burns, A.J. Wellings, Calculating controller area network (can) message response times, Control Eng. Pract. 3 (8) (1995) 1163–1169.
[45] V. Van der Veen, L. Cavallaro, H. Bos, et al., Memory errors: the past, the present, and the future, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2012, pp. 86–106.
[46] X. Wang, N. Hovakimyan, L. Sha, L1simplex: fault-tolerant control of cyber–physical systems, in: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems, ACM, 2013, pp. 41–50.
[47] Z. Wang, R. Cheng, D. Gao, Revisiting address space randomization, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2011, doi:10.1007/978-3-642-24209-0_14.
[48] R. Wartell, V. Mohan, K.W. Hamlen, Z. Lin, Binary stirring: self-randomizing instruction addresses of legacy x86 binary code, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, ACM, 2012, pp. 157–168.
[49] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, A. Sumner, Torcs, the open racing car simulator, Software available at http://torcs.sourceforge.net 4 (2000).
[50] J. Yao, X. Liu, G. Zhu, L. Sha, Netsimplex: controller fault tolerance architecture in networked control systems, IEEE Trans. Ind. Inf. 9 (1) (2013) 346–356.
[51] M.-K. Yoon, B. Liu, N. Hovakimyan, 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, ACM, 2017, pp. 143–154.

Bradley Potteiger is a PhD student in the Department of Electrical Engineering at Vanderbilt University with a research affiliation at the Institute for Software Integrated Systems. He received his MS. degree from Vanderbilt University in Electrical Engineering and his BS. degree in Computer Engineering from the University of Maryland, Baltimore County. His research at Vanderbilt is focused on Cyber Physical System (CPS) security with respect to protecting safety critical systems. Through his research he has worked with various research organizations within the government sector and industry.