Integrated Instruction Set Randomization and Control Reconfiguration for Securing Cyber-Physical Systems

Bradley Potteiger
Vanderbilt University
Nashville, TN
bradley.d.potteiger@vanderbilt.edu

Zhenkai Zhang
Vanderbilt University
Nashville, TN
zhenkai.zhang@vanderbilt.edu

Xenofon Koutsoukos
Vanderbilt University
Nashville, TN
xenofon.koutsoukos@vanderbilt.edu

ABSTRACT

Cyber-Physical Systems (CPS) have been increasingly subject to cyber-attacks including code injection attacks. Zero day attacks further exasperate the threat landscape by requiring a shift to defense in depth approaches. With the tightly coupled nature of cyber components with the physical domain, these attacks have the potential to cause significant damage if safety-critical applications such as automobiles are compromised. Moving target defense techniques such as instruction set randomization (ISR) have been commonly proposed to address these types of attacks. However, under current implementations an attack can result in system crashing which is unacceptable in CPS. As such, CPS necessitate proper control reconfiguration mechanisms to prevent a loss of availability in system operation. This paper addresses the problem of maintaining system and security properties of a CPS under attack by integrating ISR, detection, and recovery capabilities that ensure safe, reliable, and predictable system operation. Specifically, we consider the problem of detecting code injection attacks and reconfiguring the controller in real-time. The developed framework is demonstrated with an autonomous vehicle case study.

CCS CONCEPTS

• Security and privacy; **• Computer systems organization → Embedded and cyber-physical systems**;

KEYWORDS

Moving Target Defenses, Instruction Set Randomization, Cyber-Physical Systems, Resilient Architectures

ACM Reference Format:
Bradley Potteiger, Zhenkai Zhang, and Xenofon Koutsoukos. 2018. Integrated Instruction Set Randomization and Control Reconfiguration for Securing Cyber-Physical Systems. In *HoTSoS '18: Hot Topics in the Science of Security: Symposium and Bootcamp, April 10–11, 2018, Raleigh, NC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3190619.3190636

1 INTRODUCTION

In recent years there have been a number of cyber-attacks against cyber-physical systems (CPS) such as automobiles. 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 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 common cyber-attacks. This results in an increased threat from adversaries such as enemy states, and terrorist organizations who have in the past relied on physical warfare techniques to inflict damage. As such, it is important to secure these devices with the goal of limiting the attack surface and vulnerability, ensuring resiliency and reliability, and minimizing physical damage resulting from compromise.

One example vulnerability in CPS is a buffer overflow. CPS software often utilizes low level languages like C and C++ which provide a lot of flexibility in accessing memory. With often limited security mechanisms for protecting against invalid memory accessing or storing, functions like strcpy() provide the potential for input longer than a dedicated buffer size to overflow into adjacent memory locations and potentially effect program execution. As such, buffer overflows are a large scope vulnerability created from the utilization of these low level languages in CPS that allow attackers to take advantage of software input vulnerabilities to compromise running processes with malformed input. Combined with other vulnerabilities such as missing stack protections, and known software architecture frequently encountered in embedded devices, attackers can implement code injection attacks to hijack program control flow and achieve adversary code execution. In CPS such as automobiles, this amounts to allowing adversaries to potentially control the whole physical system remotely, altering acceleration, steering, and braking without needing previous or current physical access to the car.

It is important to maintain reliable, safe, and predictable operation of automobiles. Instruction Set Randomization (ISR) is a reliable moving target defense technique for protecting against code injection attacks. However, if an attack fails, but still results in the CPS controller terminating due to a kernel exception, the adversary is still able to accomplish their goal of disrupting normal operation of the system. Specifically, a failed attack attempt can lead to a Denial of Service (DoS) attack, which can be just as detrimental to the system as adversary code execution. ISR and MTD by itself is not a new problem. However, the main problem arises in how do we utilize control reconfiguration in conjunction with defense mechanisms such as ISR to maintain system availability, and real time behavior in the event of code injection attack events. The main hypothesis considered in this paper is that by using ISR, we can

HoTSoS '18, April 10–11, 2018, Raleigh, NC, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6455-3/18/04.
https://doi.org/10.1145/3190619.3190636

detect code injection attacks and reconfigure the controller fast enough to maintain safe, and reliable system behavior.

We developed a three stage control architecture consisting of attack protection, detection, and recovery. The security foundation of this architecture is based on a software ISR implementation utilizing dynamic binary translation to randomize instructions at load time, and derandomize instructions as they are fetched in the pipeline at runtime. Detection capabilities are integrated to leverage event triggered kernel POSIX signals for identifying instances of attacks. Finally, a fault tolerant recovery algorithm is integrated for transitioning between redundant software implementations in the event of an attack. The main challenge in CPS is to protect system integrity, while maintaining system availability with safe and reliable operation during a cyber-attack. Our paper makes the following contributions:

- We introduce a novel security architecture for CPS that creates a cyber-attack resistant platform using moving target defense techniques to protect against code injection.
- We implement our security architecture on a developed hardware in the loop testbed prototype using a combination of off-the-shelf embedded computing hardware and open source simulation software.
- 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 overhead presented to the system.

The rest of the paper is organized as follows: Section 2 introduces the problem of developing a MTD architecture to maintain security and system CPS properties while under attack, Section 3 presents an overview of the concepts behind our MTD architecture, Section 4 presents the system implementation of our MTD architecture, Section 5 utilizes an autonomous vehicle case study to demonstrate our MTD architecture, Section 6 presents and addresses limitations of our architecture, Section 7 presents related work and background information, and Section 8 provides concluding remarks.

2 PROBLEM FORMULATION

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. A modern car runs approximately 100 million lines of code on 50 to 70 electronic control units (ECUs) [8]. This makes a modern car a "computer on wheels," potentially subject to traditional hacking methods once thought of as only applicable to computers and information technology systems.

The internal automotive networks consist of a series of multiple communication buses with varying protocols. The most common communication busses 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 [30]. Once an attacker connects to the internal CAN Bus through vulnerabilities in any of the external communication bus interfaces, they can interact with any connected ECU in the car. This opens up automobiles to remote exploitation from adversaries. Researchers have demonstrated the applicability of buffer overflow based exploits [9], as well as performing dangerous vehicle manipulations through remote attacks [19, 20].

Code injection attacks consist of exploiting vulnerabilities in software through buffer overflow, dangling pointers, or unknown vulnerabilities to inject malicious code and divert program control flow to the malicious code for potentially causing catastrophic consequences [22]. A successful code injection attack requires system knowledge gained through reconnaissance efforts. One notable necessary system property includes the instruction set architecture. ISR is a MTD technique that can be used to defend against many types of code injection attacks [6, 13]. ISR randomizes the instruction encodings of a running program such that each process appears to have its own unique instruction set. Therefore, a piece of injected foreign code will have a very small possibility of being represented in the right format and the execution of the injected code will raise a hardware exception due to attempting to execute an invalid instruction or accessing an illegal address. This consequently leaves previous reconnaissance knowledge obsolete.

The attack model for this paper focuses on a code injection attack on a vehicle network. An example system model consists of a camera connected to a vehicle controller. The network lacks authentication mechanisms, making communication vulnerable to message spoofing. There is a buffer overflow vulnerability in the camera input processing function of the vehicle controller, combined with a lack of stack protections creating the opportunity for unbounded, executable input on the vehicle controller stack. The attack consists of an adversary gaining access to the vehicle network and spoofing malicious camera data to the vehicle controller. For the attack to be successful three assumptions are made. First, the attacker has knowledge of the system architecture necessary to craft an accurate payload. Second, the attacker has knowledge of the beginning address of the buffer input on the stack. Third, the attacker has knowledge of the relative memory location of the function return address from the beginning of the input buffer. These assumptions can be reasonably accomplished through normal reconnaissance efforts. After this knowledge is gained, the attacker crafts an input payload consisting of executable code that opens a remote terminal shell. The hexadecimal representation of the input buffer beginning address is inserted at the end of the payload in such a manner to overwrite the function return address of the vehicle controller camera input processing function. When the vehicle controller processes the malicious camera input, the payload is stored on the stack. After the function is finished, program execution redirects to the beginning of the input buffer where the payload code executes and opens up a remote bash shell to the adversary.

The main hypothesis considered in this paper is that by using ISR, we can detect and reconfigure the controller fast enough to ensure safety and stability is maintained in respect to the physical dynamics and cyber components of a CPS. To validate this hypothesis we develop a three stage security and control architecture consisting of protection, detection, and control reconfiguration capabilities. We implement this architecture on a customized hardware-in-the-loop testbed resembling the ECU setup in an automobile deployment environment. We then evaluate the architecture implementation with an autonomous vehicle case study to determine the defense

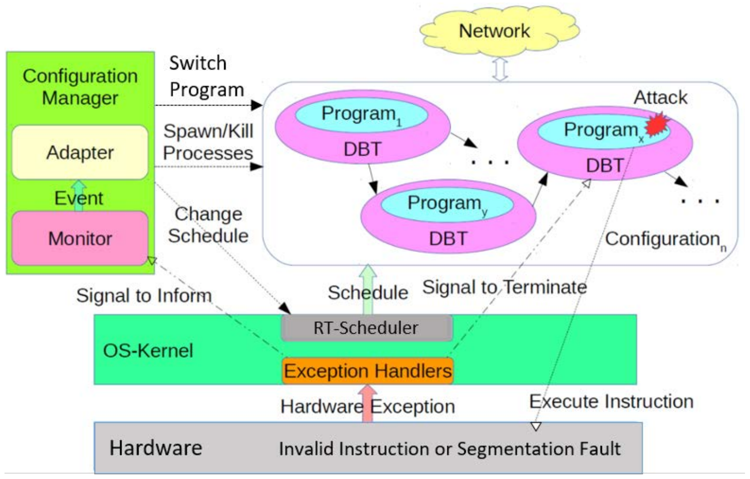

Figure 1: Control Architecture

effectiveness, as well as the ramifications on real-time performance, and physical behavior. Finally, we analyze limitations of our security architecture to identify future directions for our work.

In the rest of this paper we discuss a security architecture aimed at preventing the code injection techniques presented in our attack model, while keeping unique CPS parameters in tact. There are three objectives of our security architecture including:

(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 system crashing, even when experiencing a cyber-attack can result in devastating consequences.

3 ARCHITECTURE

The increasing attack surface facing modern CPS including zero day exploits makes it impossible to completely harden systems externally. Therefore, security has to be implemented at all levels resulting in a security in depth design. The main idea proposed in this paper is that of a software architecture utilizing a fault tolerant diversity approach with multiple redundant CPS controllers. Each controller utilizes randomization techniques at the instruction layer to hinder a successful code injection attack, while the architecture as a whole provides control reconfiguration mechanisms for recovering and switching execution between each controller. When specifically addressing code injection attacks, it is not just feasible to ignore malformed instructions, since by the time an instruction is executed the adversary has already altered the control flow of the program. As such, control reconfiguration enables the continuation of the program by recovering to a safe state.

Figure 1 presents an overview of our security architecture. The key components in the architecture are the (1) Configuration Manager that oversees, customizes, and adjusts the operation of the various operating components, (2) CPS Controllers which control the physical plant, (3) Dynamic Binary Translation Enclosure which provides the runtime environment for each CPS controller to execute various MTD techniques, (4) Operating System Kernel which handles the task scheduling and exception detection, and (5) Network which serves as the means of communication between the controllers and the distributed CPS environment. These components are described below.

Configuration Manager: This process corresponds to the high level parent process of our security architecture. Its main purpose is to spawn individual controller processes and oversee the operation of components within the architecture. This includes monitoring each CPS controller process for the occurrence of a cyber-attack, as well as specifying when and what controller to recover to in the event of a cyber-attack. Additionally, this component has the responsibility of determining the state of each CPS controller process consisting of either a running or waiting state. The Configuration Manager has signal exception handlers that are tied to exception events detected by the operating system kernel. This prevents controller crashing from being undetected. The Configuration Manager is open source so detection, and recovery algorithms can be customized to satisfy application domain specifications.

CPS Controller: This component is the actual software that controls the CPS application. In the example of automobiles, the controller takes in sensor input from the physical plant (steering, speed, breaking, camera image, etc.) and based on a developed algorithm (PID control, Neural Network, Waypoint Control, etc.) outputs an actuation command to change the physical state (change steering, increase/decrease speed). The controller implementations will vary based on the specific application domain and scenario. Each CPS controller includes subscriber functionality to receive sensor input, publisher functionality to transmit actuation output, and algorithm functionality to implement customized control logic.

Dynamic Binary Translation (DBT) Enclosure: This component is responsible for providing a randomization backend for each spawned CPS controller in the architecture. As such, this process serves as the virtual layer enclosure that manages the runtime environment of each controller component, including the ability to intercept and manipulate instructions as they are fetched before they reach the processor. Each Dynamic Binary Translation Enclosure instance can be customized from the Configuration Manager to include options to implement ISR, as well as the ability to specify an encryption key to use. If no randomization key is specified, a key will automatically be generated by the enclosure.

Operating System Kernel: This component is responsible for scheduling the underlying tasks in the architecture. Additionally, the operating system has built in exception signals that are generated when various faults occur in the system environment. The two signals utilized by the architecture include the invalid address and invalid instruction execution exceptions which both can occur when a code injection attack is attempted. The Configuration Manager has exception handlers to tie these exception events to control reconfiguration actions.

Network: This component is responsible for providing the means of communication between the CPS controller components and the rest of the distributed CPS environment. This includes receiving sensor data from the physical plant as well as transmitting actuation commands to effect physical behavior.

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. Our security architecture is designed with the goal of keeping the CPS controller from becoming compromised by the attacker.

4 SYSTEM IMPLEMENTATION

For our security architecture implementation, two stages need to be considered. These stages include design time, and runtime. This process is shown in Figure 2.

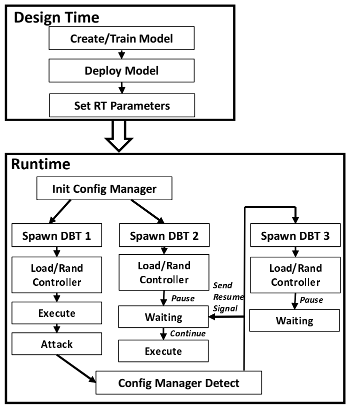

Figure 2: Architecture Process Flow

4.1 Design Time

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. In respect to real time operation, this stage consists of conducting an execution time and physical operation analysis to determine optimal sampling periods to aim for that maintain safe operation of the physical system. In respect to the autonomous vehicle case study discussed in Section 5, a single Neural Network controller exists on the hard disk. Originally, the Neural Network model is created and trained on 8 hours of driving data using a Python Machine Learning prototyping library (Keras). After the model is sufficiently trained, the model is exported to a C++ deployment environment (Tensorflow) where performance overhead is limited and execution times are minimized. At this point, after a thorough evaluation of various vehicle controller execution periods, a deadline of 100 milliseconds is found to be necessary for maintaining fast enough reaction time for the vehicle to maintain position on the road. The scheduler in our architecture uses a rate monotonic algorithm, however other techniques can be utilized such as time triggered, and earliest deadline first. As such, the period of 100 milliseconds is fed into the rate monotonic scheduler to determine the respective CPS controller task priorities.

4.2 Runtime Environment

The basis behind our security architecture involves the concept of MTD, as well as redundant executing CPS controllers with diverse randomization keys. MTD consists of changing different parameters of an environment with the goal of decreasing the probability of a successful cyber-attack. This technique is helpful in protecting large, complex systems of systems instead of starting from the beginning and investing a large amount of resources to design a flawless system from scratch. For example, if a vulnerability currently exists that allows an attacker to successfully exploit a system, periodically randomizing a system will ensure that the attacker will not be able to exploit the same vulnerability in the future. The MTD technique used in our security architecture is ISR. Utilizing Dynamic Binary Translation Enclosures, our architecture provides the capability for randomizing CPS controller executables with 32 bit encryption keys. This process is completed by XORing the text section of the ELF executable file with an encryption key. With the 32 bit randomness we are able to establish a 2^{32} degree of entropy within our system. By adjusting each CPS controller's runtime environment, DBTs create a virtual layer between the application and processor to derandomize individual instructions with a second XOR command as they are fetched before they reach the processor. A customized version of the MAMBO DBT environment [11] is utilized for providing the ISR runtime environment for CPS controllers in our security architecture. Redundant CPS controllers are included in the architecture to maintain system availability when suffering from a cyber-attack, while different randomization keys will aim to decrease the probability of the adversary having correct knowledge about the instruction architecture.

At startup, the Configuration Manager process is first loaded in memory and started. The Configuration Manager spawns two redundant child processes (MAMBO DBT Enclosure) with the CPS controller executable as input. The Configuration Manager then stores the process IDs of the DBT enclosure processes in a controller table, based on the order of startup. The first child MAMBO process started is established as executing the default CPS controller, while the second child MAMBO process started is established as executing the backup CPS controller. The Configuration Manager then sends a SIGSTOP POSIX signal to the MAMBO process controlling the backup controller to maintain a waiting state while the default controller remains in a running state. When the Mambo DBT enclosure is started, the input executable (CPS controller) is loaded in the underling Mambo process memory, a 32 bit randomization key is generated, and the controller executable is randomized with the generated key in 32 bit block increments. At this point, the loaded CPS controller is randomized and ready for runtime. This DBT start process is run for both the default and backup controller.

When looking at a snapshot at a given point in time during runtime, the default CPS controller operates under the MAMBO DBT enclosure. Program execution iterates through the randomized CPS controller instructions stored in the MAMBO process memory. As each instruction is fetched it is derandomized by performing a XOR operation with the randomization key and stored in a basic block data structure. Once a control redirection instruction is reached, the basic block is executed, registers are updated, and program execution changes to the resulting location of the program redirection command.

The Configuration Manager includes attack detection algorithms. At the basis of these algorithms are the exception handling mechanisms built into the operating system that interact between the kernel and hardware. An exception handler event driven function is included in the Configuration Manager that is called when an exception signal is received from the kernel due to various unsuspected behavior in the controller applications. For example, when a code injection attack is executed and fails, an invalid address or invalid instruction exception occurs. These exception then trigger the exception handler in the Configuration Manager to indicate that there is dangerous behavior occurring.

After an attack is detected, the recovery process begins by the Configuration Manager transitioning primary execution to the backup controller. This consists of looking up the process ID of the backup CPS controller in the controller table. Afterwards, a SIGCONTINUE POSIX signal is sent to the MAMBO DBT enclosure process of the backup controller to run the backup controller. This process is then switched to be indicated as the default controller in the Configuration Manager controller table. Afterwards, a new CPS controller is spawned under a MAMBO DBT enclosure process in the place of the attacked controller. The process ID is then inserted into the controller table as the new backup controller, and a SIGSTOP POSIX signal is sent to switch the process to a waiting state. This controller will have a new randomization key and will serve as the new backup controller in the event that the currently executing controller is attacked. Our security architecture aims to minimize the time between attack detection, and attack recovery. A maximum of no more than one full period should be missed by the system. During this time, the actuator will remain idle, but will resume normal operation at the second period when commands are received by the backup CPS controller.

5 EVALUATION

5.1 Case Study

For evaluation purposes, an autonomous vehicle case study is utilized to demonstrate the capabilities of the 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 automotive system is comprised of electronic control units controlling steering, and speed actuation, while receiving forward facing camera images as input. A neural network is utilized for controlling the steering in the autonomous vehicle based on input camera images. However, our security architecture is generic, meaning that it is not just limited to neural network controllers, and any other controller software process can be utilized instead. 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 [1]. 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 66×200 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.

5.2 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.

5.2.1 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 [3]. 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 [10] 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 3.

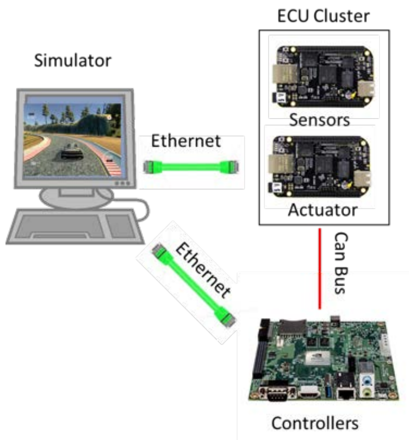

Figure 3: Testbed Hardware Architecture

5.2.2 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 simulatior.

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 [2]. 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 [11]. 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.

5.3 Attack Scenarios

There are several interesting attack vectors against autonomous vehicles including adversarial attacks against neural networks, spoofing attacks, and timing attacks. However, remotely executing code on internal automotive ECU's poses a significant threat to automotive safety and security and is addressed in this paper. The following attack scenarios are built upon the code injection attack technique discussed in Section 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.

5.3.1 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.

5.3.2 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.

5.4 Results

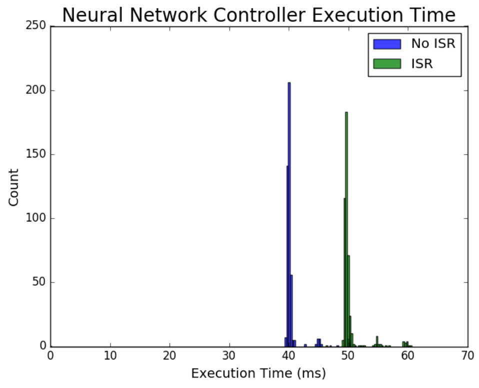

Figure 4: Neural Network Controller Execution Times

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 4, 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 5. The amount of missed deadlines during recovery is illustrated in Figure 6.

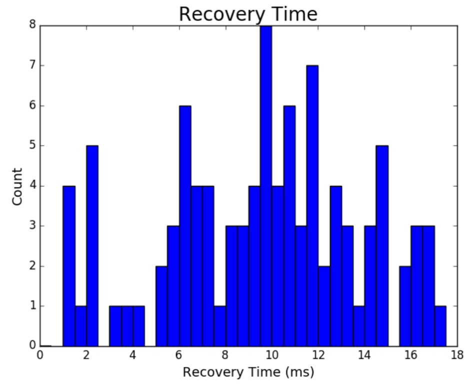

Figure 5: Recovery Times

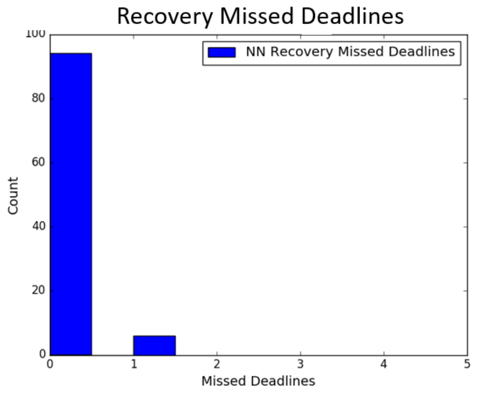

Figure 6: Recovery Missed Deadlines

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 7.

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.

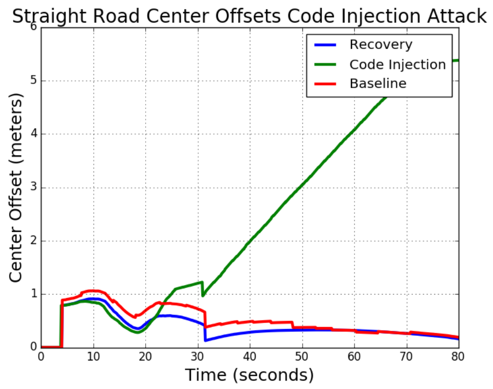

Figure 7: Code Injection Attack Straight Road Center Offset Time Plot

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 8.

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.

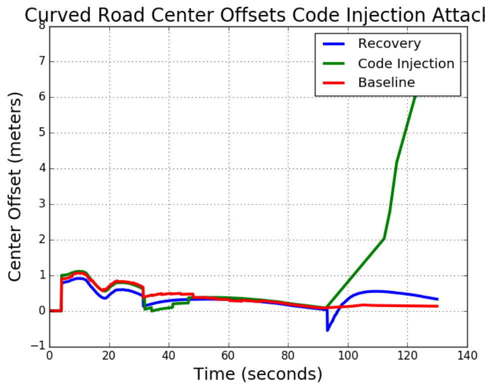

Figure 8: Code Injection Attack Curved Road Center Offset Time Plot

6 LIMITATIONS

In the following section, a few limitations of our current security architecture are discussed, as well as our plans to address them in future versions. These limitations include potential vulnerability to denial of service, code reuse, non control data, and side channel attacks.

During normal control reconfiguration under our architecture when a cyber-attack is detected, execution is transferred to a backup redundant CPS controller, and the Configuration Manager spawns a new CPS controller to serve as the new backup controller and take the place of the crashed controller. However, this process consists of a minimum amount of recovery time before the backup controller is fully up, missing at most 1 deadline. If an attacker were to rapidly execute multiple cyber-attacks, the constant reconfiguration process would potentially result in denial of service behavior due to missing deadlines. We address this problem by implementing a last resort fail safe controller that executes in the case of a rapid reconfiguration attempt. In the scenario of the autonomous vehicle case study, this consists of stopping the vehicle on the road, and waiting for the system to be brought up again by the operators.

Our current security architecture is designed to mitigate against code injection attacks which rely on inserting system compliant executable instruction payloads through software vulnerabilities. The technique of ISR deals well with this attack by changing the runtime instruction environment as to render the attackers injected instruction format invalid. However, this technique does not prevent against code reuse or non control data attacks which are also threats to the CPS domain. To address these cyber attacks, we plan on implementing address space randomization, and data space randomization techniques in the future version of our architecture. Address space randomization will change the layout of functions in memory to prevent an attacker from gaining knowledge to redirect to a specific function in a code reuse attack. Additionally, data space randomization will encrypt the stored data variables in the program to protect against non control data attacks. Furthermore, low entropy versions of ISR such as 32 bit implementations have been shown to be potentially defeated through plain text leakage, and side channel analysis. By integrating periodic dynamic reconfiguration in the future, the variation of the runtime randomization key will lower the accuracy of attacker reconnaissance knowledge gained through these methods.

Side channel attacks are large threats to CPS applications by enabling adversaries to reverse engineer encryption keys and running instructions through analyzing physical attributes. In the current version of our architecture we randomize each CPS controller with a different randomization key at load time. In the absence of a cyber attack, the randomization key of the running CPS controller will remain static, suscepting the program to side channel attacks. To address this potential vulnerability we plan on implementing periodic dynamic randomization key changing in the future version of our security architecture.

7 RELATED WORK

Previous work relating to automobile security includes securing the CAN protocol communication [16], implementing data encryption, authentication, honeypots, and intrusion detection systems [15], and securing communication between external devices and the vehicle network [12]. These techniques rely on the designer having full knowledge of vulnerabilities, as well as the correct locations of where to place defense mechanisms. It has been commonly shown that researchers and adversaries can find ways around defense mechanisms in place, whether due to a lack of sufficient protections, or by introducing zero day exploits not previously considered [20]. As such, CPS need to focus on backup resiliency mechanisms to ensure that if an attack such as code injection can defeat current defense mechanisms, the attack will not effect the system integrity, or availability. It is the authors' view that ISR is a beneficial technique that is best utilized in combination with current defense mechanisms, and other MTD techniques to lessen the probability of a successful cyber-attack. However, the reconfiguration aspect of this paper is vital to the resiliency of the system in responding to cyber-attacks, and faults with the goal of maintaining CPS integrity and availability.

ISR implementations can be either hardware or software based. Software based ISR implementations are based on using dynamic binary modification (DBM). DBM can be accomplished by using either dynamic binary translator (DBT) tools, or customizing software emulators. In DBTs such as MAMBO [11], STRATA [26], and PIN [17], each instruction can be altered (derandomized) as it is fetched before it reaches the processor. Therefore, a program can be statically randomized before runtime with a key, and derandomized at runtime as instructions are fetched before processor execution. [23] describes an ISR implementation using DBT tools while [7, 14] describes ISR implementations using software emulators. ISR can also be implemented using hardware by using FPGA's or customized processors. Researchers used the OpenSPARC FPGA processor to create a hardware based ISR prototype [29].

To recover from attacks, multiple techniques have been used in the area of software fault tolerance. One of the more common fault tolerance techniques is diversity, which consists of using N-Version programming to generate multiple different software versions that satisfy the same functionality specification [4]. Due to the independent generation structure, each version will use a different algorithm whose vulnerability set is independent, decreasing the probability of the same vulnerability occurring in multiple software versions. As such, this technique is built off of the assumption that if one software version fails, the other versions will not fail under the same constraints. Recovery blocks are a common technique to recover between software versions. Recovery blocks involve creating a checkpoint before the current version is executed, and attempting to try alternative versions after failure, resuming at the respective saved state [18, 24, 25].

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 [28]. 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 [21], Net Simplex [32], and L1 Simplex [31]. 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 [27], pacemakers [5], and unmanned aerial vehicles [33].

8 CONCLUSION

In this work we've shown how ISR can be leveraged to protect against code injection attacks in a CPS such as an autonomous vehicle. Due to the safety-critical nature of autonomous vehicles, it is important for the system to remain in a constant reliable and operable state, even while under attack. As such, we investigated recovery mechanisms and fault tolerant methodologies to detect an attack and recover to a backup controller while minimizing the probability of a subsequent similar attack due to N-Version programming. A software moving target defense architecture implementation was developed to protect, detect, and recover from cyber-attacks. We described the architecture including the randomization framework and control manager process. Additionally we described a hardware in the loop testbed that we developed for the purposes of evaluating our architecture with an autonomous vehicle case study. By implementing our architecture in the hardware in the loop testbed we were able to obtain live measurements of our controller execution and recovery times, as well as analyze the impact that the cyber-attacks had on the physical dynamics of the vehicle driving behavior. It was shown that our security architecture had limited overhead on the execution times of the CPS controllers, and by recovering in a rapid manner, the vehicle was able to successfully transfer to a backup controller while minimizing the amount of missed deadlines and effect on the driving behavior. This work is the first part of developing a comprehensive moving target defense security architecture for safety-critical CPS. In the future, our work will look into integrating our architecture with address space randomization, data space randomization techniques, and dynamic reconfiguration as well as developing and formalizing metrics for establishing the impact of security mechanisms in CPS.

9 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 (70-NANB15H263). 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] End-to-end deep learning for self-driving cars. https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/. (Accessed on 06/04/2017).

[2] 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).

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

[4] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[5] 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*, pages 99–107. IEEE, 2009.

[6] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[7] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289. ACM, 2003.

[8] R. N. Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[9] 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.

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

[11] 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)*, 13(1):14, 2016.

[12] K. Han, A. Weimerskirch, and K. G. Shin. Automotive cybersecurity for in-vehicle communication. In *IQT QUARTERLY*, volume 6, pages 22–25, 2014.

[13] 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*, pages 2–12. ACM, 2006.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[15] 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*, pages 528–533. IEEE, 2011.

[16] 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*, pages 1–7. IEEE, 2012.

[17] 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*, volume 40, pages 190–200. ACM, 2005.

[18] M. R. Lyu. *Software fault tolerance*. John Wiley & Sons, Inc., 1995.

[19] C. Miller and C. Valasek. Adventures in automotive networks and control units. *DEF CON*, 21:260–264, 2013.

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

[21] 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*, pages 65–74. ACM, 2013.

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

[23] 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*, pages 41–48. ACM, 2010.

[24] L. L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, 2001.

[25] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, (2):220–232, 1975.

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

[27] 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). Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.

[28] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

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

[30] 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*, pages 1–12. IEEE, 2013.

[31] 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*, pages 41–50. ACM, 2013.

[32] J. Yao, X. Liu, G. Zhu, and L. Sha. Netsimplex: Controller fault tolerance architecture in networked control systems. *IEEE Transactions on Industrial Informatics*, 9(1):346–356, 2013.

[33] 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*, pages 143–154. ACM, 2017.