2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)

Security in Mixed Time and Event Triggered Cyber-Physical Systems using Moving Target Defense

Bradley Potteiger, Feiyang Cai,
Abhishek Dubey, Xenofon Koutsoukos
Vanderbilt University
Nashville, TN

Zhenkai Zhang
Texas Tech University
Lubbock, TX

***Abstract*—Memory corruption attacks such as code injection, code reuse, and non-control data attacks have become widely popular for compromising safety-critical Cyber-Physical Systems (CPS). Moving target defense (MTD) techniques such as instruction set randomization (ISR), address space randomization (ASR), and data space randomization (DSR) can be used to protect systems against such attacks. CPS often use time-triggered architectures to guarantee predictable and reliable operation. MTD techniques can cause time delays with unpredictable behavior. To protect CPS against memory corruption attacks, MTD techniques can be implemented in a mixed time and event-triggered architecture that provides capabilities for maintaining safety and availability during an attack. This paper presents a mixed time and event-triggered MTD security approach based on the ARINC 653 architecture that provides predictable and reliable operation during normal operation and rapid detection and reconfiguration upon detection of attacks. We leverage a hardware-in-the-loop testbed and an advanced emergency braking system (AEBS) case study to show the effectiveness of our approach.**

***Index Terms*—Moving Target Defense, Time Triggered, Event Triggered, Cyber-Physical Systems**

I. Introduction

Safety-critical CPS often contain memory corruption vulnerabilities such as buffer overflows that allow for the remote exploitation of software. Cyber-attacks such as code injection, code reuse, and non-control data attacks allow for adversaries to hijack safety-critical functionality, potentially resulting in severe damage to the system and the surrounding environment. Memory corruption attacks pose a serious threat since they allow adversaries to remotely control and alter sensor data or execute unsafe actuation behavior while making it look like normal behavior to monitoring subsystems.

Instruction set randomization (ISR) [17], [36], address space randomization (ASR) [35], and data space randomization (DSR) [4] are all very effective moving target defense techniques (MTD) that mitigate these types of attacks [25]. However, when disrupting the attack process, these techniques result in the software crashing due to an exception such as an invalid instruction, invalid address, or unsafe data value, thus resulting in a denial of service which is not acceptable in CPS.

Safety-critical CPS often utilize time-triggered architectures to ensure predictable and reliable operation [13]. However, in the event of an attack, it is necessary to respond as fast as possible instead of waiting until the following period of the static schedule. As such event triggered functionality is necessary to provide rapid detection, and reconfiguration at the point of the attack. By combining time triggered and event triggered functionality into a mixed approach, the predictability benefits of time triggered systems, and the rapid response benefits of event triggered systems can be maintained.

This paper presents an approach to protect against code injection, code reuse, and non-control data attacks by developing a mixed time and event-triggered MTD security architecture. Further, we consider how to limit the overhead of the approach and we explain how to reconfigure the system upon detection of an attack to limit the amount of missed deadlines. Our hypothesis is that by integrating ISR, ASR, and DSR with the ARINC 653 standard [31], we can protect against code injection, code reuse, and non-control data attacks while rapidly performing reconfiguration to maintain system safety. The contributions of this paper are as follows:

- We develop and implement a mixed time and event-triggered MTD security architecture that provides predictable and reliable operation during normal circumstances as well as rapid detection of attacks and reconfiguration to maintain safety. The security architecture includes ISR, ASR, and DSR to protect against code injection, code reuse, and non-control data attacks. Furthermore, we leverage the benefits of the ARINC 653 standard such as isolation, static schedule creation, and health monitoring for our architecture.
- We design and develop a novel reconfiguration scheme to maintain CPS safety and static schedule integrity during a cyber-attack.
- We implement the architecture using a hardware-in-the-loop testbed representative of modern CPS to

2375-5261/20/$31.00 ©2020 IEEE
DOI 10.1109/ISORC49007.2020.00022

evaluate the approach. Our implementation includes detection and recovery capabilities to limit missed deadlines and maintain safe and reliable operation.
- We present an autonomous vehicle case study to demonstrate the effectiveness in limiting the impact of attacks in the context of an advanced emergency braking system (AEBS).

The rest of this paper is outlined as follows: Section II discusses the key background fundamentals behind our approach, Section III introduces the threat model utilized as motivation for our paper, Section IV provides an overview of our security approach and implementation, Section V illustrates the evaluation of our architecture through the use of an autonomous vehicle case study, Section VI discusses related work, and Section VII ends the paper with concluding remarks.

II. Background

A. ARINC 653

ARINC 653 is a popular standard implemented for aerospace and other safety-critical CPS applications aimed at maintaining safety and predictability within critical components [11]. As this standard forms the foundation of the proposed architecture, it is useful to describe the basic components. ARINC 653 is comprised of two types of components: partitions, and processes. These components have various types of attributes, states, and communication methods that can be configured throughout the design process. These components are briefly described below.

Partitions are the highest level of abstraction and include a shared memory space running tasks. Partitions can essentially be considered as a virtual machine in a single application environment comprising of data, context, and attributes. In a partition, there can exist multiple processes that correspond to the actual tasks in an application. It is easier to think of the partition in terms of a virtual machine, and a process as a specific software application running inside of the virtual machine. As such, multiple processes can run concurrently and rely on the operating system for scheduling.

Communication within ARINC 653 is conducted through the use of unidirectional channels, ports, and messages. Additionally, communication is possible both between partitions (inter-partition) as well as within a partition between multiple processes (intra-partition). It is important to note that the data transmitted is considered an atomic entity meaning that if the whole message is not received then no part of the message will be received.

Finally, ARINC 653 contains a health monitor that analyzes the behavior of the underlying partitions and processes for anomalies. In our approach, the health monitor is also responsible for the reconfiguration process of child processes when a cyber-attack is detected.

B. Moving Target Defense Techniques

The MTD techniques used in our architecture include ISR [17], ASR [3], and DSR [4]. Legacy CPS software often contains numerous memory corruption vulnerabilities such as buffer overflows that allow attackers to remotely perform code injection, code reuse, and non-control data attacks [6]. By randomizing the internal structure of software, attacker reconnaissance efforts are ineffective, resulting in failed cyber-attack attempts.

In a code injection attack, adversaries leverage a memory corruption vulnerability to inject and execute code remotely on the program stack [26]. To successfully execute code, the injected instructions format must be consistent with the native system processor format (x86, ARM, etc.). By randomizing the representation of native instructions at runtime using ISR, attacker injected code will be of an invalid representation resulting in an invalid instruction exception.

In a code reuse attack, adversaries leverage a memory corruption vulnerability to divert control flow to another location within the program memory such as a safety-critical function [32]. To be successful, attackers must know the memory location of their target to divert program control flow effectively. By randomizing the address layout using ASR, the location of target functions will no longer be as expected by the attacker, resulting in diverting control flow to the wrong location. Hence, any code reuse attack attempts will result in an invalid memory access exception.

In a non-control data attack, adversaries leverage a memory corruption vulnerability to overwrite adjacent safety-critical variables [29]. To be successful, attackers must know the variable format, allowing them to correctly alter the variable value. With DSR, the variable representations will be randomized, and any attacker manipulation will result in a value wildly different than expected. This increases the ease of detecting any malicious data tampering activity, preventing the attacker from successfully overwriting critical variables.

C. Control Reconfiguration

It is not enough to detect and stop cyber-attacks in CPS where it is also required to maintain safe operation. Availability is a key property that can be maintained using reconfiguration. A popular approach to control reconfiguration in safety-critical CPS is the simplex architecture [24]. Simplex contains two controllers: a default controller for normal execution, and a safety controller that serves as a backup in case of failure in the default controller. The default controller is designed to be high performance, while potentially containing vulnerabilities. The safety controller, on the other hand, may not provide optimal performance but guarantees safe, and secure operation. Additionally, Simplex includes a decision module that determines when to switch between the two controllers and perform the execution transitioning process.

III. Threat Model

The threat model considers memory corruption attacks such as code injection, code reuse, and non-control data attacks on a vehicle network. An example attack vector consists of the adversary compromising the telematics control unit (TCU) through the remote cellular interface and pivoting to hijack the remote function actuator (RFA). With access to a direct communication channel with the driving controller, the adversary can craft a message payload to take advantage of the memory corruption vulnerability and alter control. At this point, an attacker can perform an attack where they can leverage the buffer overflow to affect safety-critical behavior in the driving controller.

The following assumptions are made in the proposed security architecture. First, the sensor and actuator clusters are fully secure. The driving controller electronic control unit (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 knows the relative address of a safety-critical variable relative to the start of the input buffer. Finally, the attacker knows the underlying software architecture of the safety-critical controllers, allowing them to target the most impactful variables and functions. These assumptions are not impractical given examples demonstrated in the literature [23].

It is important to note that our threat model makes the assumption that the only vulnerable component is the driving controller (CPS controller). Other components in our architecture such as partitions and the health monitor are assumed to be secure, and only the driving controller partition will include MTD defense protections. The first assumption is valid because ARINC 653 contains one-way communication constraints [11] that make it impossible for the attacker to pivot into any other partition from the driving controller. With regards to the second assumption, CPS software is normally legacy code without modern day compiler security protections. To add to this, source code is normally unavailable, meaning that it becomes very difficult to verify the lack of vulnerabilities within the driving controller. In comparison, the health monitor is a relatively simple program containing a few hundred lines of C++ code. Since the source code is readily available, and compiled from source by the designer, security vulnerabilities can be identified and coding best practices can be established before the deployment of the architecture. This assumption can be justified by the requirement of highest level of certification required for components that monitor the health of system [16]. As such, within our architecture we can assume that the designer has previously identified and patched vulnerabilities within the health monitor, making that component secure.

IV. Security Architecture

A. Components

The key components in our approach are: (1) CPS Controllers which control the physical plant, (2) Dynamic Binary Translator (DBT) which uniquely customizes the runtime environment for each CPS controller, (3) points to analysis graph (PAG) which describes the relationship between pointers and variables within a program, and (4) Health Monitor which controls the reconfiguration upon detection of an attack. The components are described below.

1) CPS Controllers: This component is the actual software that controls the CPS application. The controller receives sensor inputs from the system, performs computation operations, and outputs actuation commands. Our architecture supports a broad array of control techniques and applications.

2) Dynamic Binary Translator (DBT): This component is responsible for providing a unique randomization backend for each spawned CPS controller in the architecture. In other words, the DBT is a virtual sandbox layer that serves as an intermediary between the executing binary and the processor. The DBT intercepts instructions as they are fetched and alter program semantics before execution by the processor. The open-source instrumentation tool Mambo [12] is utilized to support the DBT implementation.

3) Points-To-Analysis: This component is responsible for using static analysis techniques to identify the variable relationships within a program. By feeding this information into the DSR implementation, we can identify correct randomization keys to utilize for various memory locations.

4) Health Monitor: This component is responsible for detecting an attack and rapidly reconfiguring the system to spawn backup controllers to take over functionality and minimize safety-critical component downtime. Additionally, the Health Monitor is responsible for executing the static time-triggered schedule. This component is a requirement specified in the ARINC 653 standards [11].

We assume that the DBT and the Health Monitor are not susceptible to cyber-attacks. Therefore, the variable key storage table in each DBT is assumed to be secure against integrity attacks.

B. Design Time

1) Component Selection: For component selection, it is important to consider several properties including memory usage, slack time, and deadlines. The security architecture introduces some overhead in both memory usage and performance. For safety-critical components that have strict and tight deadlines, a comprehensive assessment is required to determine if there is enough flexibility within the current implementation to support the introduced overhead. Furthermore, the approach provides

several combinations of MTD security protections allowing designers to optimize the trade-off between security and performance.

2) Time-Triggered Design: ARINC 653 allows us to distribute the application into separate, isolated partitions executing in sequential order. In our design, we first have to perform an execution analysis of the relevant system components identifying the maximum time required for the processes to complete. After this step, we build in slack time and assign the required time allotments for each partition in the system. The assigned time allotment must be larger than the maximum execution time of the underlying process. Otherwise, system processes may not fully finish, and the behavior of the system could be consequentially affected. For our implementation we leverage the open source ARINC 653 software emulator [1].

3) Moving Target Defense (MTD): In the ISR implementation, we randomize instructions with a 32-bit key dynamically generated at runtime, creating a high degree of entropy protection. For the ASR implementation, we shuffle functions and randomize base memory addresses, significantly decreasing the probability of success of code reuse and return-oriented programming attacks. Finally, for the DSR implementation, we XOR stack-based variables with a 64-bit key, while also utilizing a redundancy comparison check for determining the presence of a non-control data attack.

Both ISR and ASR have different degrees of granularity to optimize the trade-off between security and performance. ISR can use different types of randomization based on XOR or AES 256 encryption. Furthermore, different memory ranges can be randomized with different keys, reducing the likelihood of the adversary correctly guessing the randomization key. ASR by default is defined with coarse-grained granularity meaning that the base addresses of the program, stack, heap, and shared libraries are different for every runtime instance. However, fine-grained granularity is built-in meaning that not only the base addresses are unique but function locations are shuffled as well.

4) Lift Target Binary: For static analysis, it is optimal to convert the binary program into an intermediate representation (IR) format. The low-level virtual machine (LLVM) compiler bit code is utilized for this purpose [19]. To convert a native binary to LLVM bit code, we utilize Binary Ninja for disassembly and control flow recovery [5] and Mcsema for IR instruction translation [8], [22].

5) Points-To-Analysis: In a program, an object can either be an instruction or a memory location. Points-To-Analysis is utilized to produce the associations between instructions and the memory locations that they access through load and store instructions. By recording the associations between memory locations and instructions, we can create a map corresponding to what randomization keys to utilize for respective program instructions. The Points-To-Analysis process produces a Points-To-Analysis Graph (PAG) as output which allows for identifying the relationships between instructions and memory locations. For our implementation we leverage the SVF library [37].

C. Runtime

The runtime environment is integrated into the DBT component which encapsulates the vulnerable CPS controller as shown in Figure. 1. There are two inputs to the DBT: the binary executable itself and a text file defining the instruction and memory associations from the PAG. Once these inputs are received by the DBT, the binary will be randomized at load time with the ISR, ASR, and DSR randomization modules while being derandomized as instructions are accessed with the runtime module.

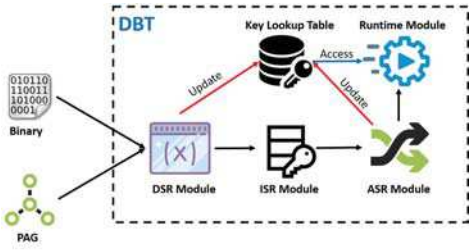


Fig. 1: MTD Initialization Process

During the randomization process, a 32-bit key is dynamically generated and each instruction in the program is randomized through an XOR operation. After this step, the binary is pushed through the ASR module where the functions are shuffled. Since after this step, the program instructions have different addresses compared with the program after DSR, we adjust the DSR table with the updated addresses of the respective load/store instructions. Finally, we start the execution of the newly randomized binary. After every instruction is fetched, it is derandomized by performing another XOR operation before being decoded by the pipeline. Anytime there is a store instruction a randomization instruction is inserted which XORs the value with the respective associated key from the lookup table and stores the variable at the appropriate location on the stack. Additionally, a duplicate copy with a duplicate randomization key is also stored in an adjacent location to the newly randomized variable. When a load instruction is encountered, both copies of the variable are loaded and derandomized with their respective keys found from the lookup table. After derandomization, the plain text values are compared for equivalence. If both values are equal then the program can proceed as normal. However, if the values are different, an "Attacked Variable" exception is generated and the program is terminated (Figure 2).

D. Control Reconfiguration

For reconfiguration, we leverage the Simplex architecture [24]. Our reconfiguration scheme is triggered when an attacker attempts to perform a code injection, code reuse, or non-control data attack. Since the attack vector has been moved due to MTD, any attack attempt results

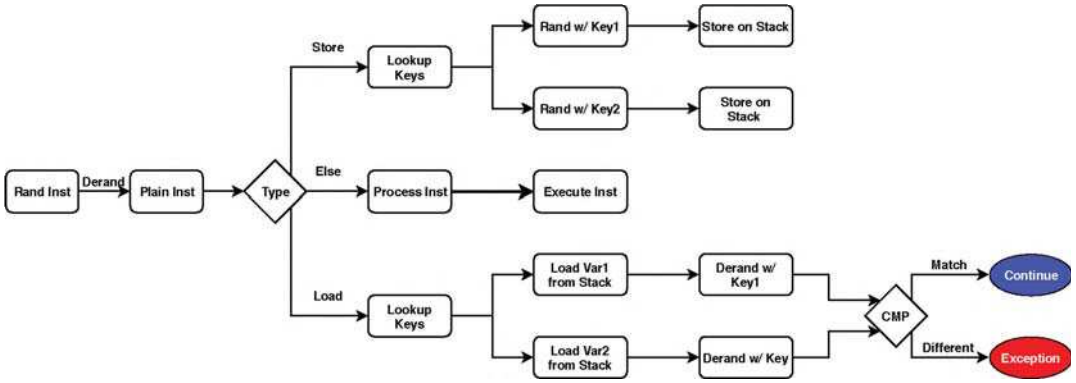


Fig. 2: MTD Runtime Instruction Pipeline

in an invalid instruction, invalid memory access, or invalid variable comparison exception respectively. This exception is then detected from a signal handler within the Health Monitor component which triggers the reconfiguration process, transitioning program execution to the backup controller [10], [21]. Based on our threat model, the only step where this process can fail is during the execution of the backup controller. As such, we make the assumption that there are no bugs in the backup controller.

V. Evaluation

The case study in this paper is based on an Advanced Emergency Braking System (AEBS) which safely and comfortably stops a host vehicle to avoid collision with a lead car. The automotive system contains ECUs that receive sensor measurements such as camera images, speed and send actuation commands such as braking, steering, and throttle. A convolutional neural network is used for perception to compute an estimated distance to the lead vehicle. Then a braking controller feed-forward neural network, and a PID throttle controller are used to compute the actuation signals. Both the controller and perception neural networks are created using the Tensorflow Lite library [1]. We consider a scenario where the lead vehicle brakes at a stoplight, thus requiring the host vehicle's AEBS system to be activated. The goal is to brake and avoid a collision. The system is illustrated in Fig. 3.

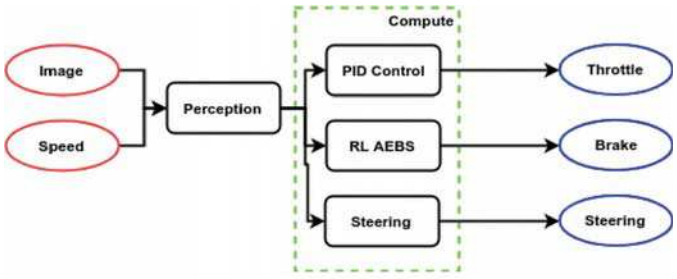


Fig. 3: AEBS Case Study

[1]https://www.tensorflow.org/lite

A. Attack Scenario

The host vehicle contains several ECUs for the various components. External interfaces that include cellular communications from the TCU for remote monitoring services and RFID sync with the vehicle key fob can be used for deploying cyber-attacks. The driving controller constantly polls for the key fob signal to determine if the engine should remain on. Since the TCU is connected remotely through a cellular interface, this component is at risk of compromise. The attacker can exploit the TCU, pivot to the RFA, and then transmit malicious packets to the driving controller, exploiting a buffer overflow vulnerability in the input processing function.

Once the adversary injects malicious input into the driving controller, we consider 3 attack scenarios. First, a code injection attack can occur where the attacker injects a malicious payload to open a remote shell, disabling the driving controller and starting a malicious controller to fully accelerate into the lead car. Second, a code reuse attack can occur where the attacker utilizes the buffer overflow to divert control flow to a steering function within the driving controller software, causing the vehicle to turn left off of the road. Third, a non-control data attack can occur where the attacker utilizes the buffer overflow to overwrite an adjacent safety-critical variable (distance to the lead vehicle). At this point, the AEBS algorithm will believe that the lead vehicle is further ahead than its real location, resulting in a collision.

B. Experiment Setup

To evaluate the impact of cyber-attacks, we develop a hardware-in-the-loop testbed. The testbed includes embedded hardware representing typical CPS infrastructure and a simulation workstation to represent the vehicle and the physical environment. The architecture of the testbed provides the capability to implement real-time CPS control algorithms to interact with and operate an autonomous car in a connected simulator.

1) Autonomous Vehicle Simulator:: The autonomous vehicle simulator used in our testbed is the CARLA autonomous vehicle simulator [9]. In the testbed, the

simulator runs on Ubuntu 18.04. Socket-based communication is provided to access variables in the simulation. We also use a customized python API interface for easing variable access from external processes. The simulator can be customized to output sensor data such as lidar, speed, images, distance to objects, orientation, and GPS locations. Actuation input can change variables that affect steering, acceleration, and braking.

2) CPS Controllers:: The software for the controllers is executed on an NVIDIA Jetson TX2 board. The board is configured with the Linux4Tegra 28.2 operating system, GPU libraries such as CUDA, and machine learning libraries such as Tensorflow.

3) Communication:: Communication between the simulator NVIDIA Jetson TX2 board is implemented via Ethernet and the ZeroMQ (ZMQ) communication library [2].

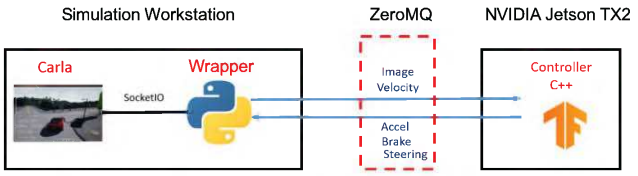


Fig. 4: Testbed Setup

C. ARINC 653

We have 4 partitions comprising the static schedule: sensor data receipt, perception, computation, and actuation transmission. The sensor partition is responsible for receiving the updated image, and speed data from the CARLA simulator. The perception partition which contains a process that computes an estimated distance based on the provided images. Then, the processing partition which contains a computation process determines an optimal throttle and braking value the neural network and PID controllers. Finally, the output command is fed to the actuation partition which transmits the requested command to the CARLA simulator. It is important to note that the only vulnerable partition in this setup is the processing partition due to its interaction with the external facing ECUs within the automotive network. As such, we harden all processes within this partition with our MTD security protections.

1) Static Schedule: To accurately establish the static schedule, we must conduct execution time analysis to determine the appropriate allocated amount of time for each partition. Too much allocated time can result in inefficiency in the approach while too little allocated time can result in system failure. To identify an upper bound of the execution time of each process, we record 100,000 iterations under varying conditions. With this data, we can identify the average, as well as outlier values. After performing the execution time analysis, we identify the statistics for each process shown in Table I.

[2]https://zeromq.org/

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

TABLE I: Execution Time Analysis

Based on the execution analysis, we use the static schedule shown in Figure 5. The superframe period is defined to be 100ms and will repeat continuously throughout the system's lifetime. This period is small enough to support the functionality of the vehicle but also provides enough slack time to support various system processes.

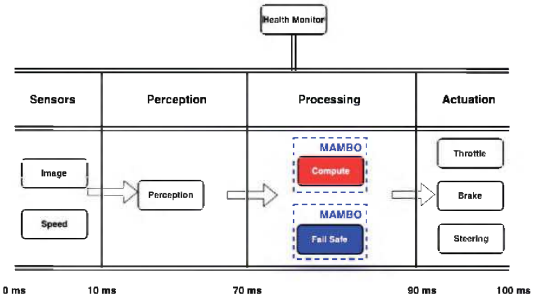
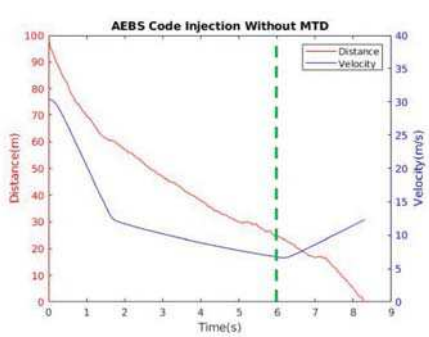


Fig. 5: Static Schedule

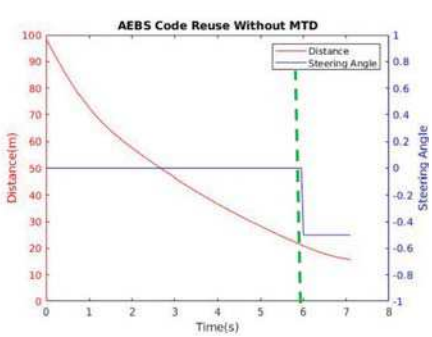
To verify that our static schedule is correct, we need to analyze worst-case scenarios by determining if the aperiodic attack detection and recovery processes executed by the Health Monitor fit within the critical slots of the Partition 3 schedule. If this is shown to be true, then the designed schedule is guaranteed to be fully schedulable [14]. In the designed schedule, the default controller is in Partition 3 and the recovery processes and fail-safe controller are triggered by an attack. This means that at the earliest, the Health Monitor attack detector will be triggered from the beginning of Partition 3 to the end of Partition 3. Since the default controller is the only initial process in Partition 3, the Health Monitor and fail-safe controller processes will have full access to the CPU for the remainder of partition 3. However, as a worst-case scenario, we consider the case of the attack occurring at the end of the Partition 3 time allotment. In this case, the fail-safe controller will not complete its execution until a full period later during the next time allotment of Partition 3 and during the current period, the previous actuation value is used.

D. Results

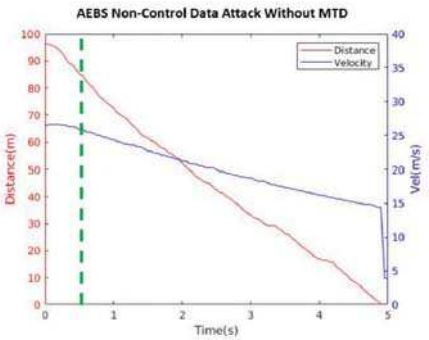
1) Static Analysis: For the case study a 3 layer Neural Network is used for the AEBS controller and a PID component is responsible for speed and steering control. However, these two components are negligible in size so will focus our efforts mainly on the neural network performance. In



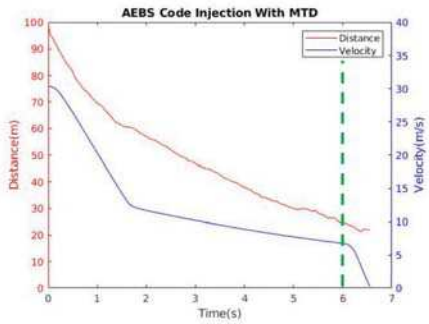
a: Code Injection Attack/no MTD



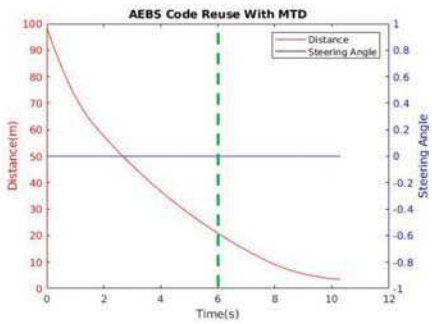
b: Code Reuse Attack/no MTD



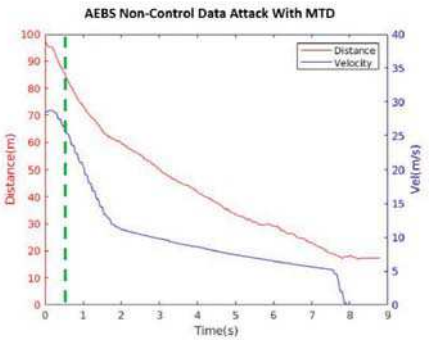
c: Non-control Data Attack/no MTD



d: Code Injection Attack/with MTD



e: Code Reuse Attack/with MTD



f: Non-Control Data Attack/with MTD

the implementation, there are 1025 variables with a file size of approximately 220 Kilobytes. Additionally, there are two shared libraries that we need to secure: Tensorflow Lite, and libm.

The first stage in the static analysis pipeline is binary lifting, averaging approximately 17 milliseconds of execution time for 1000 executions. The second stage is points-to-analysis. To evaluate the scalability of the points-to-analysis implementation, we run 100 iterations of generating PAGs, averaging execution times of approximately $250ms$.

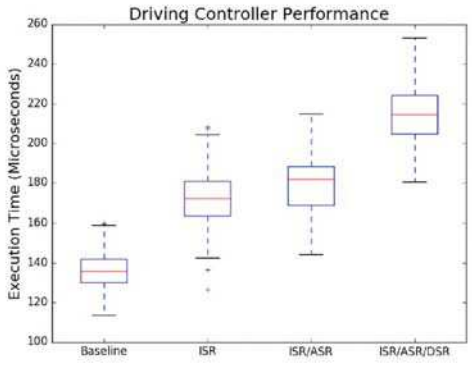


Fig. 7: Controller Execution Times

2) Runtime Performance: In our scenarios, we have three combinations enabled including one with only ISR, a second with ISR and ASR, and a final combination with all three MTD techniques enabled. The results are illustrated. in Figure 3. With only ISR enabled, there is an overhead of approximately 28% while with ISR and ASR there is a little higher overhead at approximately 31%. However, with the significant increase in overhead created by DSR, the final combination with all three techniques enables results in overhead at approximately 59%. Although this performance overhead can be acceptable, in cases where significant overhead is unacceptable, performing a risk assessment is necessary to determine the optimal combination of MTD techniques. With the introduced overhead and complexity, it is important to ensure that the system model is adjusted and verified at design time to maintain safety and functionality of the system. Furthermore, the designer has to ensure that there is enough slack time allocated to handle the reconfiguration necessary during a cyber-attack.

During a code injection attack, a malicious payload is injected to spawn a remotely accessible root shell within the vehicle operating system. The attacker will then terminate the default controller and spawn a malicious controller that will fully accelerate the vehicle in a straight path. At this point, as can be observed in Figure 6a, the vehicle speeds up and crashes into the back of the lead vehicle. However, with ISR enabled, the instructions of the controller are randomized resulting in an inaccurate instruction format in the payload that will cause an invalid instruction execution exception. After that, the system

switches to a backup controller which fully brakes the vehicle. Figure 6d shows that the system can successfully recover to the backup failsafe controller in time to brake the vehicle before crashing into the lead car.

During a code reuse attack, the attacker leverages a buffer overflow vulnerability to redirect control-flow to an existing turn left function implemented using the steering controller. At this point, control flow executes a left turn resulting in the vehicle turning left (Figure 6b). However, with ASR enabled, the memory layout of the control software is different, resulting in the function no longer exist in the target memory address, and consequently leading in an invalid memory access exception. At this point, recovery transitions execution to the backup failsafe controller where the car brakes before unsafe behavior (Figure 6e).

During the case of a non-control data attack, the adversary can manipulate the controller operation by altering the perceived distance to the lead vehicle. With this adjustment, the new distance value is set to $100m$ causing the host vehicle to maintain its speed and crash into the lead car as illustrated in Figure 6c. However, with DSR and variable integrity checking enabled, the attempt by the attacker to overwrite the distance variable will result in an incorrect variable comparison consequently flagging the attack. At this point, a failsafe controller takes over execution and fully brakes the vehicle. As a result, safety is preserved and the host vehicle avoids a collision as shown in Figure 6f.

3) Attacks Prevented: MTD techniques, in general, are designed to limit the ability of adversaries to collect accurate reconnaissance knowledge on a system, consequently failing to craft a valid exploit. The security approach is designed specifically to protect against any stack-based remote injection attack. The architecture protects against buffer overflow based exploits, including code injection, code reuse, and non-control data attacks. However, our approach also has the potential to protect against other vulnerabilities such as heap overflows, integer overflows, and dangling pointers. There are limiting factors for the applicability. For example, when the attacker has direct access to system program execution, denial of service attacks will result in constant reconfiguration.

VI. Related Work

Moving target defense implementations have traditionally been independent with ISR including both hardware [27] and software versions [17], ASR including coarse grained [20] and fine grained versions [7], and DSR including source code [4] and IR implementations [29]. Additionally, control reconfiguration algorithms such as Simplex have normally focused on the aspect of fault tolerance with regard to maintaining the safety of CPS [33]. Our work over the last couple of years has built upon these two principles by showing the viability of MTD integration with control reconfiguration to support security while ensuring the reliable operation of the respective safety-critical CPS [28], [30].

With regards to time triggered implementations within the literature, work has focused on the obfuscation of the static schedule, randomizing the order of tasks to prevent reconnaissance against application secrets [18], [41]. Additionally, software defined networking techniques such as Openflow [15] have become popular to mitigate against the interception of communication and targeting of hosts. We propose that all of these techniques are complementary, integrating defenses at different layers of abstraction to provide comprehensive protection against a maximal amount of attacks. Our approach complements this work by providing protections at the application layer, mitigating against software related exploits that can lead to the hijacking of safety-critical controllers.

Simplex, which is the primary motivator of our security architecture, has been a widely utilized fault tolerant architecture [34]. Several previous simplex based implementations include Secure System Simplex [24], Net Simplex [39], and L1 Simplex [38]. Furthermore, simplex architectures have been popular in safety-critical applications such as flight control systems [33], medical devices [2], and unmanned aerial vehicles [40].

VII. Conclusion

In this paper, we have shown how ISR, ASR, and DSR can be integrated to support protections against code injection, code reuse, and non-control data attacks in the context of safety-critical CPS applications. The MTD architecture was successfully used in a mixed time-triggered and event-triggered architecture to support predictable operation during normal circumstances while maintaining rapid detection and reconfiguration during a cyber-attack. Finally, by developing a hardware-in-the-loop testbed, we can demonstrate the approach in a realistic setting. Experimentation produced positive security protections against all three classes of attacks considered. Also, we were able to recover to failsafe control rapidly. In conclusion, the proposed MTD approach can be used for CPS runtime environments that are resilient to buffer overflow based cyber-attacks.

VIII. Acknowledgements

This work is supported in part by the National Security Agency (H98230-18-D-0010), the National Science Foundation (CNS-1739328), and by NIST (70NANB18H198). 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 NSA, NSF, or NIST.

References

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

[2] 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.

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

[4] S. Bhatkar and R. Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[5] M. Capelletti. Unlinker: an approach to identify original compilation units in stripped binaries. *POLITesi*, 2017.

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

[7] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies*, 2016(4):454–469, 2016.

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

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

[10] A. Dubey, W. Emfinger, A. Gokhale, P. Kumar, D. McDermet, T. Bapty, and G. Karsai. Enabling strong isolation for distributed real-time applications in edge computing scenarios. *IEEE Aerospace and Electronic Systems Magazine*, 34(7):32–45, 2019.

[11] A. Dubey, G. Karsai, and N. Mahadevan. A component model for hard real-time systems: Ccm with arinc-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011.

[12] 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.

[13] G. Heiner and T. Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, pages 402–407. IEEE, 1998.

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

[15] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132, 2012.

[16] L. A. Johnson et al. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, 199, 1998.

[17] 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, 2003.

[18] K. Krüger, G. Fohler, and M. Volp. Improving security for time-triggered real-time systems against timing inference based attacks by schedule obfuscation. 2017.

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

[20] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291, 2015.

[21] N. Mahadevan, A. Dubey, and G. Karsai. Application of software health management techniques. In *Proceedings of the 6th international symposium on software engineering for adaptive and self-managing systems*, pages 1–10. ACM, 2011.

[22] F. Markl. Case study on llvm as suitable intermediate language for binary analysis. *ret*, 32:0.

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

[24] 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.

[25] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, and W. Streilein. Survey of cyber moving target techniques. Technical report, MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, 2013.

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

[27] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. Asist: architectural support for instruction set randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 981–992, 2013.

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

[29] B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated data space randomization and control reconfiguration for securing cyber-physical systems. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, page 3. ACM, 2019.

[30] B. D. Potteiger. *A Moving Target Defense Approach Towards Security and Resilience in Cyber-Physical Systems*. PhD thesis, 2019.

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

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

[33] 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.

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

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

[36] A. N. Sovarel, D. Evans, and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *USENIX Security Symposium*, volume 10, 2005.

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

[38] 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.

[39] 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.

[40] 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.

[41] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.