Journal of Systems Architecture 125 (2022) 102420

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Moving target defense for the security and resilience of mixed time and event triggered cyber–physical systems

Bradley Potteiger [a,*], Abhishek Dubey [b], Feiyang Cai [b], Xenofon Koutsoukos [b], Zhenkai Zhang [c]

[a] Johns Hopkins Applied Physics Lab, Laurel, MD, United States of America
[b] Vanderbilt University, Nashville, TN, United States of America
[c] Texas Tech University, Lubbock, TX, United States of America

ARTICLE INFO

Keywords:
Moving target defense
Time triggered
Event triggered
Cyber–physical systems

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.

1. 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) [1,2], address space randomization (ASR) [3], and data space randomization (DSR) [4] are all very effective moving target defense techniques (MTD) that mitigate these types of attacks [5]. 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 [6]. 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.

Mixed time-triggered and event triggered functionality is often implemented through the use of partitioned operating systems, which ensure that the tasks being triggered across separate temporal boundaries do not impact each other. Operating Systems such as LynxOS-178 [7] and Tresos [8] have been used in the avionics and automotive domains. One of the side affects of using the partitioned system concept, codified in the ARINC-653 standard [9], is the ability to launch applications belonging to different criticality levels and security domains. The partitioned operating systems also include a comprehensive health management architecture that can handle errors at the level of the application as well as temporal groupings called partitions.

While partitioned architectures provide robust protection against known faults, they are not necessarily robust against security threats that may be present in the applications as shown by recent enhancements to the standard [10]. In this paper we study this problem and provide a robust approach to protect against code injection, code reuse, and non-control data attacks for such partitioned systems using a MTD security architecture. Specifically, we look at how to integrate MTD

* Corresponding author.
E-mail address: bpottei1@umbc.edu (B. Potteiger).

https://doi.org/10.1016/j.sysarc.2022.102420
Received 7 October 2020; Received in revised form 16 January 2022; Accepted 20 January 2022
Available online 5 February 2022
1383-7621/© 2022 Published by Elsevier B.V.

techniques with the ARINC 653 architecture to support additional security protections within mixed time triggered and event triggered systems. 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 (previously explored MTD techniques used in isolation) with the ARINC 653 standard [9], we can protect against code injection, code reuse, and non-control data attacks while rapidly performing reconfiguration to maintain system safety. This paper is an extension of our previous work in [11]. 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 present an extended threat model and present how our developed architecture presents a comprehensive defense-in-depth approach against a broad array of attacks.
- 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 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). We provide a through evaluation and discussion based on domain specific simulation results, computation execution times, and reconfiguration times.

The rest of this paper is outlined as follows: Section 2 introduces the threat model utilized as motivation for our paper, Section 3 provides an overview of the system architecture, Section 4 presents the security architecture of our approach, Section 5 illustrates the evaluation of our architecture through the use of an autonomous vehicle case study, Section 6 provides a discussion of our architecture security capabilities, Section 7 discusses related work, and Section 8 ends the paper with concluding remarks.

2. Threat model

The modern-day vehicle is essentially a “computer on wheels”, with dozens of electronic control units (ECUs), millions of lines of code, and hundreds of vulnerabilities. Furthermore, there are several points of entry that allow attackers to launch potentially devastating attacks remotely. Once a foothold is obtained, attackers can have free reign to manipulate actuation, and potentially divert control to unsafe behavior.

An exemplary vehicle system model that our threat model is based on includes 6 components: a sensor cluster, actuator cluster, driving controller, telematics control unit (TCU), remote function actuator (RFA), and RFID sensor (Fig. 1). The sensor cluster provides critical data representing the current state of the vehicle such as the speed, and front facing camera image. The actuator cluster provides the ability to manipulate the vehicular acceleration through the throttle and brake, and vehicle angle through steering. The driving controller is responsible for performing computation based on the provided sensor cluster input, and outputting commands to the actuation cluster. In this paper, the driving controller is an AEBS controller that is responsible for braking the vehicle to avoid colliding with upcoming objects. Both the TCU, and

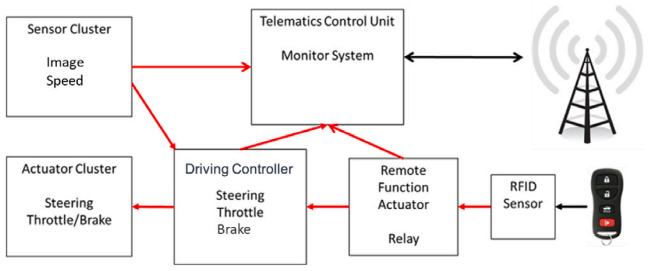


Fig. 1. Example vehicle system model.

RFA are responsible for providing the external interface for the vehicle. The TCU monitors the various metrics of the system, transmitting data to a remote operating station for maintenance and emergency purposes. The RFA is responsible for determining the presence of a key fob for allowing the vehicle to be turned on.

The threat model in this paper 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 [12].

It is important to note that our threat model makes the assumption of a trusted computing base, meaning that the only vulnerable component is the driving controller (CPS controller). Other components in our architecture such as partitions, the Dynamic Binary Translators, 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 [13] 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 [14]. 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.

To evaluate the effectiveness of our architecture within the context of an autonomous vehicle case study, we utilize a developed hardware-in-the-loop testbed. We further utilize physical metrics such as vehicle

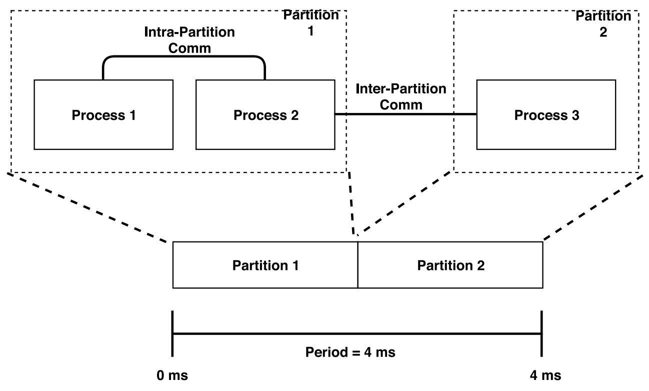


Fig. 2. Time triggered application architecture.

position combined with software metrics like performance overhead in both normal operation and attack scenarios. Finally, to conclude that our hypothesis is true two observations need to be clear from the results: (1) The performance overhead needs to be minimal enough to ensure that execution times do not exceed designed real time constraints and (2) Vehicles need to follow safe driving behavior, stopping completely before colliding with the parked vehicle on the road. In the event that both of these observations are true, we can conclude that our architecture is successful.

3. System architecture

ARINC 653, the backbone of our architecture, is a popular standard implemented for aerospace and other safety-critical CPS applications aimed at maintaining safety and predictability within critical components [13]. 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 in Fig. 2.

Partitions are the highest level of abstraction in the ARINC 653 standard and include a shared memory space running tasks. At any given time, a partition can be defined with one of several possible modes. A mode controls the behavior of the respective partition, such as whether tasks can execute, how long it will take for the partition to start up, and whether memory can be accessed. The modes that are most relied upon are the idle and normal modes.

In a partition, there exists multiple processes that correspond to the actual tasks in an application. Multiple processes can run concurrently and are reliant on the operating system for their respective scheduling capabilities. Similar to regular applications, processes can be initialized, terminated, reinitialized, and paused through control commands sent by the parent partition. At any given time, processes can be described by one of several states: dormant, ready, running, and waiting.

Communication within the ARINC 653 framework is conducted through the use of unidirectional channels, ports and messages. Additionally, communication is possible both between partitions (inter-partition), as well as within a partition between multiple processes (intra-partition).

At the backbone of the ARINC 653 standard is the temporal schedule. The temporal schedule, also referred to as a static schedule, is a sequence of consecutive partition time allotments that build up to create one large periodic time allotment called the hyper-period. The partitions themselves are constrained to only execute within their specific time allotment within the overall hyper-period. As such, at any given time this schedule is deterministic, improving the reliability of the overall system.

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

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 of child processes when a cyber-attack is detected. The health management architecture in ARINC 653 provides prescriptive lookup tables and an ordered mitigation plan for handling errors at the level of the applications, partitions and the whole module (a collection of partitions). These tables are setup during system initialization. Every process when it detects an error can raise an alert that is successively passed through the levels and handled as specified. An example of the architecture implementation is available in [15,16]. We use this architecture in this paper.

4. MTD architecture

The MTD techniques used in our architecture include ISR [1], ASR [17], 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 [18]. 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 [19]. 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 [20]. 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 [21]. 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.

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 [22]. 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.

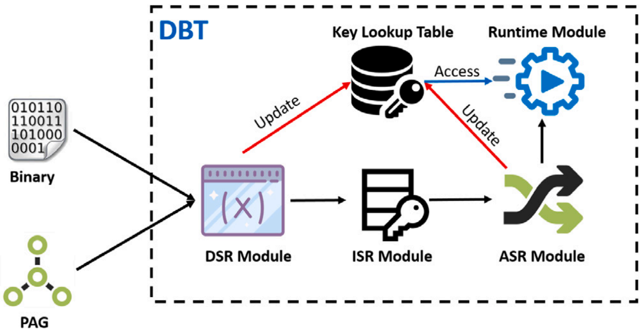


Fig. 3. MTD initialization process.

4.1. 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.

4.1.1. 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.

4.1.2. Dynamic Binary Translator (DBT)

This component is responsible for providing a unique randomization backend for each spawned CPS controller in the architecture. In other words, the DBT is a virtual sandbox layer that serves as an intermediary between the executing binary and the processor. The DBT intercepts instructions as they are fetched and alter program semantics before execution by the processor. The open-source instrumentation tool Mambo [23] is utilized to support the DBT implementation.

4.1.3. 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.

4.1.4. 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 [13].

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.

4.2. Design time

4.2.1. 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.

4.2.2. 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 [24].

4.2.3. 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.

4.2.4. 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 [25]. To convert a native binary to LLVM bit code, we utilize Binary Ninja for disassembly and control flow recovery [26] and Mcsema for IR instruction translation [27,28].

4.2.5. 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 [29].

4.3. Runtime

The runtime environment is integrated into the DBT component which encapsulates the vulnerable CPS controller as shown in Fig. 3. 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

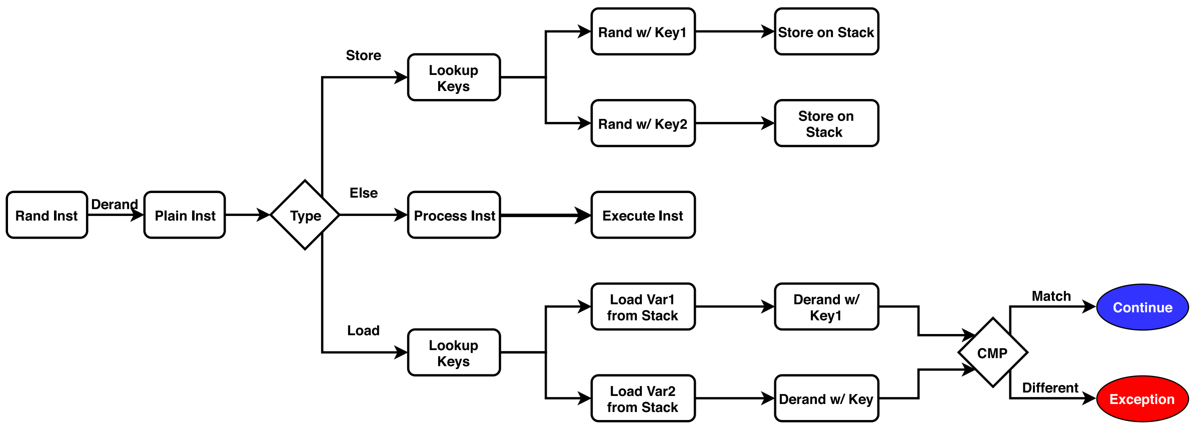


Fig. 4. MTD runtime instruction pipeline.

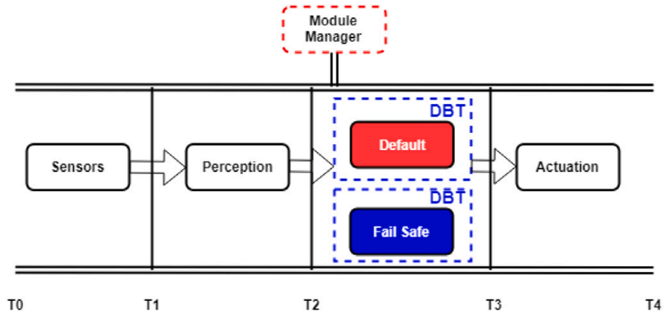


Fig. 5. Reconfiguration setup.

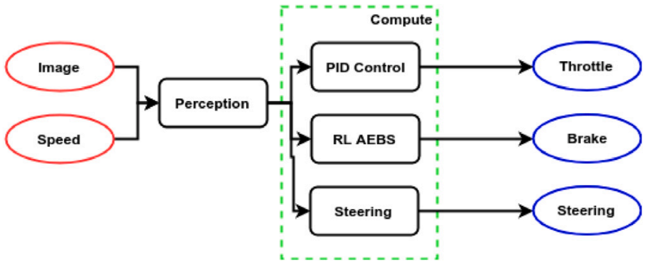


Fig. 6. AEBS case study.

modules while being derandomized as instructions are accessed with the runtime module.

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 (Fig. 4).

4.4. Control reconfiguration

For reconfiguration, we leverage the Simplex architecture [22]. 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 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 [16,30]. 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. Our reconfiguration setup is illustrated in Fig. 5.

5. 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. 6.

5.1. 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

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

Fig. 7. HIL testbed.

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.

5.2. 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. Our testbed configuration is illustrated in Fig. 7 with the specific components illustrated in Fig. 8.

5.2.1. Autonomous vehicle simulator

The autonomous vehicle simulator used in our testbed is the CARLA autonomous vehicle simulator [31]. 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.

5.2.2. 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.

5.2.3. Communication

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

[2] https://zeromq.org/.

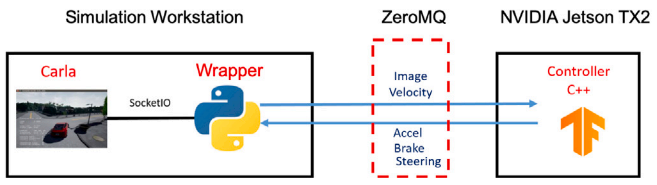


Fig. 8. Testbed setup.

Table 1
Execution time analysis.

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

5.3. ARINC 653

We have 4 partitions comprising the static schedule: sensor data receipt, state estimation, processing, and actuation transmission. The sensor partition is responsible for receiving the updated image, and speed data from the CARLA simulator. The state estimation partition contains a perception process that computes an estimated distance based on the provided images. Furthermore, the processing partition contains a computation process determines an optimal throttle and braking value the neural network and PID controllers. Finally, the actuation partition 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 (see Fig. 9).

5.3.1. 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 a reasonable estimate of the execution time of each process, we resorted to measurement-based WCET analysis, recording 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 1.

Based on the execution analysis, we use the static schedule shown in Fig. 10. The superframe period is defined to be 100 ms 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.

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 [32]. 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

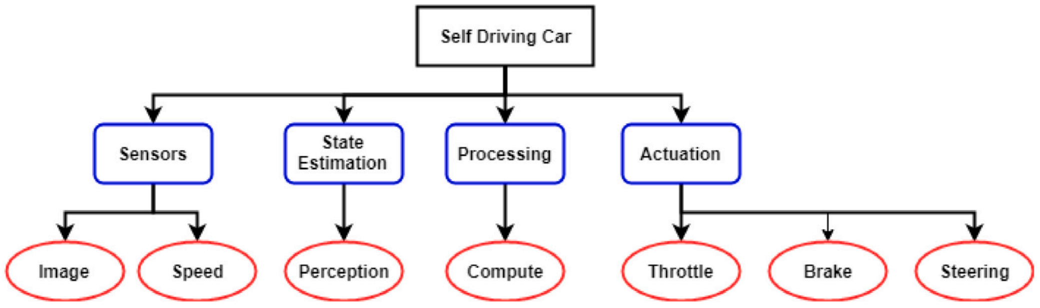


Fig. 9. Autonomous vehicle processes.

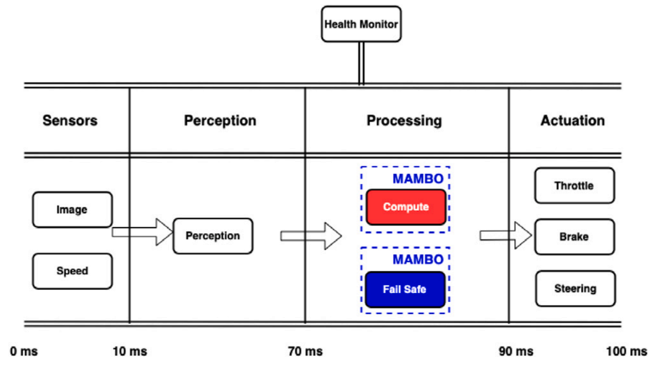


Fig. 10. Static schedule.

Table 2
ARINC 653 experiment configuration.

ARINC 653 configuration				
	Part1	Part2	Part3	Part3
# of processes	1	1	2	1
Length	10 ms	60 ms	20 ms	10 ms
Base priority	99	99	99	99
Stack size	4 kB	4 kB	4 kB	4 kB
Deadline type	Soft	Soft	Soft	Soft

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.

5.4. Results

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

5.4.1. Static analysis

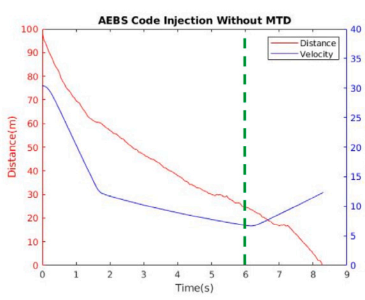
For the case study a 3 layer Neural Network is used for the AEBS controller and a PID component is responsible for speed and steering control. However, these two components are negligible in size so will focus our efforts mainly on the neural network performance. In the implementation, there are 1025 variables with a file size of approximately 220 Kilobytes. Additionally, there are two shared libraries that we need to secure: Tensorflow Lite, and libm. Static analysis in this case is performed on a combination of a desktop host and the target system.

The first stage in the static analysis pipeline is binary lifting, which is performed on a remote desktop host. This process averages approximately 17 ms 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 250 ms. This time is to an extent dependent on the target platform that static analysis is performed on, but will in effect be negligible in the overall framework due to only requiring a one time performance cost before runtime.

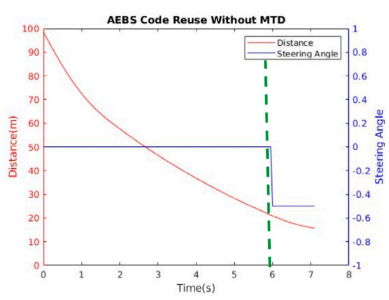
5.4.2. Runtime performance

The second stage focuses on analyzing runtime performance overhead on the target system to ensure accurate design time constraints are maintained. 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 Fig. 12. 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. Finally, the results from the system response to the attack attempts are illustrated in Fig. 11. The dashed lines in the subfigures represent the point in time when the cyber attack occurs, conveying the respective system response to the code injection, code reuse, and non-control data attacks.

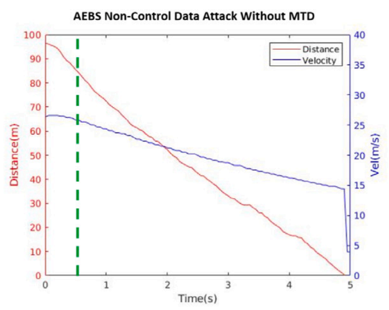
During a code injection attack, a malicious payload is injected to spawn a remotely accessible root shell within the vehicle operating system. The attacker will then terminate the default controller and spawn a malicious controller that will fully accelerate the vehicle in a straight path. At this point, as can be observed in Fig. 11(a), 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.



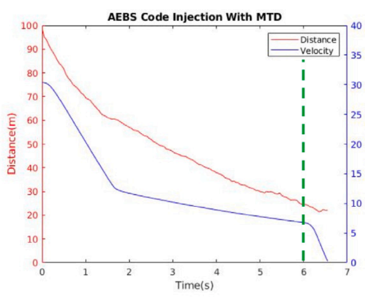
a: Code Injection Attack/no MTD



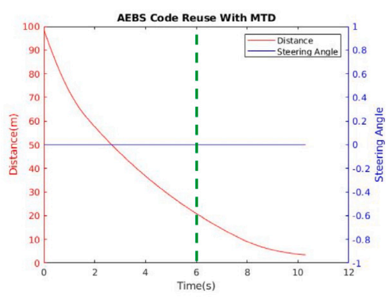
b: Code Reuse Attack/no MTD



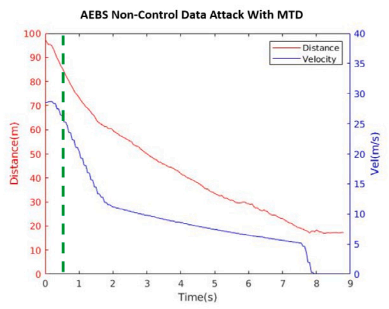
c: Non-control Data Attack/no MTD



d: Code Injection Attack/with MTD



e: Code Reuse Attack/with MTD



f: Non-Control Data Attack/with MTD

Fig. 11. Vehicle case study attack results.

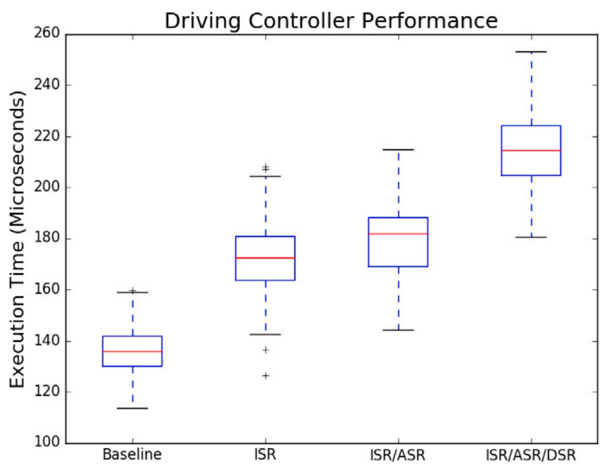


Fig. 12. Controller execution times.

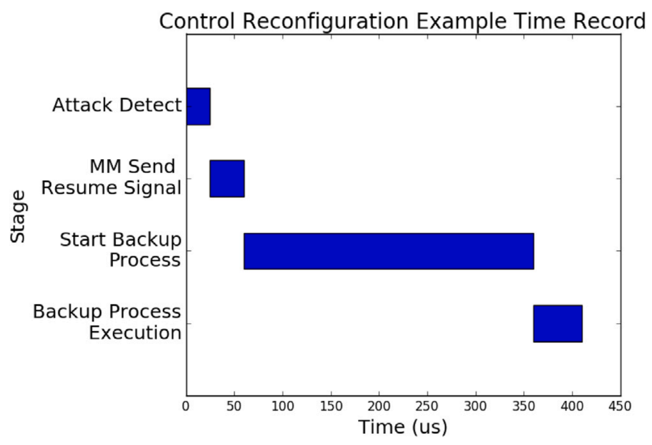


Fig. 13. Control reconfiguration example time record.

Fig. 11(d) 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 (Fig. 11(b)). 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 (Fig. 11(e)).

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 100 m causing the host vehicle to maintain its speed and crash into the lead car as illustrated in Fig. 11(c). 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 Fig. 11(f).

Finally, Fig. 13 illustrates the time record of an example reconfiguration process. As such, the recovery process consists of four possible stages: attack detection, Health Monitor send resume POSIX signal, start Backup Controller process, and execute Backup Controller to compute a new actuation value. As observed, the attack detection, and POSIX signal stages are relatively negligible while the controller start and execution times consume most of the reconfiguration time. Therefore, the reconfiguration time from attack detection to transmission of a new actuation command directly correlates to the size and execution time of the backup controller.

6. Discussion

When considering the threat model for this paper, it is important to note that there are two layers of defenses within our architecture: one builtin to the ARINC 653 architecture, and the second layer within the MTD framework. Looking at remote attacks in general, once access is obtained to a system, the adversary normally pivots between components to discover sensitive data, manipulate other processes, and generate backdoors to the system. However, due to the memory segmentation of partitions within the ARINC 653, it is necessary for attackers to leverage inter-partition communication for their pivoting exercises. Furthermore, due to the uni-directional properties of ARINC 636 inter-partition communication, it is not possible for attackers to communicate from a destination to source partition. As such, if an architecture is designed with security in mind, partitions with remote connectivity will not have inter-partition communication capabilities with partitions storing safety-critical processes, mitigating against this pivoting approach. In this case the security principles of the architecture will remain true as long as the remote interface partitions remain a "destination" of any inter-partition connection with partitions containing safety-critical processes. If this is not the case, designers have to take the assumption of compromise into account and build extra security precautions into the safety-critical partitions to mitigate against attack damage and access. To accomplish this goal, MTD techniques within our architecture are integrated to effectively mitigate against the probability of vulnerability discovery and exploit mitigation within the respective safety-critical partition.

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 in our architecture is designed specifically to protect against any stack-based remote injection attack, the most common technique leveraged for remote autonomous vehicle attacks. Our architecture specifically 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. However, these factors are limited from the underlying communication structure of the ARINC 653 architecture.

The evaluation of our architecture conveys the successful ability to disrupt remote attacks within autonomous vehicles while recovering rapidly and reliably enough to ensure the safety of three different scenarios. Furthermore, looking at the analysis, there is a significant degree of overhead that is presented with all three MTD techniques enabled. However, our architecture can be fine tuned to minimize performance overhead through the configuration of the security setup and recovery process. First, the MTD techniques ISR and DSR are the primary drivers of overhead within our security architecture, while ASR presents minimal overhead. As such, to optimize security with minimal overhead, ASR should be solely leveraged, limiting overhead to approximately 1%, a more acceptable amount to fit within existing slack time. The second configuration decision is based on the recovery process. Looking at the breakdown of the recovery process, the startup time for the backup controller takes up the majority of the overall recovery time. As such, to minimize the recovery time, the backup controller should be started with the default controller but remain in an idle state until needed. As such, this approach will involve more memory, but when recovery is needed, execution transference will be more rapid, decreasing the underlying instability of the CPS controller and improving safety and reliability.

7. Related work

Moving target defense implementations have traditionally been independent with ISR including both hardware [33] and software versions [1], ASR including coarse grained [34] and fine grained versions [35], and DSR including source code [4] and IR implementations [21]. Additionally, control reconfiguration algorithms such as Simplex have normally focused on the aspect of fault tolerance with regard to maintaining the safety of CPS [36]. 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 [37,38].

When introducing moving target defense techniques such as ISR, ASR, and DSR into existing real time software implementations, performance overhead is created that was not taken into account during the initial design of the system. As such, with the added performance overhead of security mechanisms, controller execution times can increase to the point of exceeding their previously assigned deadlines. In this case, it is imperative that the system be dynamically readjusted to adapt the associated sampling rates of underlying controllers. Generally, as the sampling rate decreases, the input-feedforward passivity index of the system decreases. In order to ensure stability and safety of the closed loop system for adaptive sampling rates, passivation methods described by input–output transformations that generalize typical methods of series, feedback and parallel (or feedforward) interconnections to passivate a system need to be considered [39–41]. The range of appropriate sampling rates needs to be defined at design time to establish a safe bounds of operation for the system.

To ensure proper real time operation of CPS, run time assurance monitoring is necessary. This involves the checking of complicated properties that involves analyzing many system variables. Thus, this procedure produces a significant overhead to the system. To reduce this overhead, a lot of previous work has focused on combining static analysis techniques with dynamic analysis runtime checking. Additionally, symbolic techniques in the compiler optimization process have been explored. The main tradeoff that must be taken into account is between monitoring accuracy, and system overhead [42]. The designer must take into account whether it is acceptable to miss occasional execution events that potentially could effect accuracy, or check every execution event for maximum accuracy, but consequently increase the probability of missing a deadline due to the higher overhead. Simplex based architectures rely on recovery mechanisms that make the assumption that the fail safe controller will be effective in all scenarios once a violation is detected by the monitoring module. To solve this problem, developed architectures have focused on incorporating diagnostic capabilities, and user specified recovery plans to determine the appropriate path to take once a violation is detected [43–45]. Another approach taken instead of recovery is called runtime enforcement which attempts to prevent potential violations by delaying and reordering events leading to various unsafe states. Runtime enforcement architectures have been proven to be effective for security applications, but rely on the ability to block events to allow for real time checking [46–48]. Runtime enforcement produces a higher overhead compared to a strictly recovery based system under normal conditions, but produces a higher level of security for the specified property violation. A proper balance between security, and overhead can be established by utilizing runtime enforcement for the most critical security violations, while relying on recovery for unexpected less critical violations.

In the past, time and event triggered architectures were generally considered mutually exclusive. However, recent research has focused on combining the predictability and reliability of time triggered architectures with the flexibility of event triggered architectures to support optimal operation of safety critical CPS [32]. This hybrid approach utilizes a time triggered periodic schedule for normal system operation, while event triggered sporadic events are reserved for less frequent event occurrences such as error detection or an aperiodic communication event. By building in enough slack into the static schedule at design time, these systems can be verified to be fully schedulable, meaning that at any point a sporadic process can execute without disrupting the timing requirements of the time triggered tasks. One of the most popular implementations of this approach is the ARINC 653 standard which is commonly found in aviation, automobile, and space designs [49–51]. In our approach we utilize a component based ARINC 653 implementation to support time triggered and event triggered capabilities in our architecture [13].

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 [52,53]. Additionally, software defined networking techniques such as Openflow [54] 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.

Redundancy has played a vital role for fault tolerance in safety-critical applications. Traditional past examples include airplanes [55], military protocols [56], and cloud servers [57]. However, the most visible recent example of redundancy is illustrated within the autonomous vehicle domain where multiple controllers, sensors, and communication busses are utilized to ensure that if one component fails, the behavior of the overall system is not compromised. Even though redundancy has traditionally been utilized for safety in high availability systems, the same fundamental properties can be applied to application security. N-version programming has long been used to mitigate compromised controller effects [58], and multithreaded based implementations have demonstrated the ability detect violations in application consensus [59]. We leverage this previous work by applying redundancy to the program local variables on the stack, inserting variable comparison checking before usage to ensure data integrity.

Simplex, which is the primary motivator of our security architecture, has been a widely utilized fault tolerant architecture [60]. Several previous simplex based implementations include Secure System Simplex [22], Net Simplex [61], and L1 Simplex [62]. Furthermore, simplex architectures have been popular in safety-critical applications such as flight control systems [36], medical devices [63], and unmanned aerial vehicles [64].

8. 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported in part by the National Security Agency, United States of America (H98230-18-D-0010), the National Science Foundation, United States of America (CNS-1739328), and by NIST, United States of America (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] G.S. Kc, A.D. Keromytis, V. Prevelakis, Countering code-injection attacks with instruction-set randomization, in: Proceedings Of The 10th ACM Conference On Computer And Communications Security, 2003, 272–280.
[2] A.N. Sovarel, D. Evans, N. Paul, Where's the FEEB? The Effectiveness of Instruction Set Randomization, in: USENIX Security Symposium, vol. 10, 2005.
[3] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, A.-R. Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization, in: 2013 IEEE Symposium on Security and Privacy, IEEE, 2013, pp. 574–588.
[4] S. Bhatkar, R. Sekar, Data space randomization, in: International Conference On Detection Of Intrusions And Malware, And Vulnerability Assessment, Springer, 2008, pp. 1–22.
[5] H. Okhravi, M. Rabe, T. Mayberry, W. Leonard, T. Hobson, D. Bigelow, W. Streilein, Survey of cyber moving target techniques, Technical report, MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, 2013.
[6] G. Heiner, 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), IEEE, 1998, pp. 402–407.
[7] LynuxWorks, RTOS for Software Certification: LynxOS-178. http://www.lynuxworks.com/rtos/rtos-178.php.
[8] Autosar GbR, AUTomotive Open System ARchitecture. http://www.autosar.org/.
[9] P.J. Prisaznuk, ARINC 653 Role in integrated modular avionics (IMA), in: 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, IEEE, 2008, pp. 1–E.
[10] A. Baker, P. Parkinson, Cyber security enhancements for a safety-critical arinc 653 avionics platform, in: Twenty-Sixth Safety-Critical Systems Symposium, 2018.
[11] B. Potteiger, F. Cai, A. Dubey, X. Koutsoukos, Z. Zhang, Security in Mixed Time and Event Triggered Cyber-Physical Systems using Moving Target Defense. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC), 89–97, 2020. http://dx.doi.org/10.1109/ISORC49007.2020.00022.
[12] C. Miller, C. Valasek, Remote exploitation of an unaltered passenger vehicle, Black Hat USA 2015 (2015).
[13] A. Dubey, G. Karsai, N. Mahadevan, A component model for hard real-time systems: CCM with ARINC-653, Softw. - Pract. Exp. 41 (12) (2011) 1517–1550.
[14] L.A. Johnson, et al., DO-178B, software considerations in airborne systems and equipment certification, Crosstalk 199 (1998).
[15] A. Dubey, G. Karsai, N. Mahadevan, A component model for hard real-time systems: CCM with ARINC-653, Softw. - Pract. Exp. 41 (12) (2011) 1517–1550, http://dx.doi.org/10.1002/spe.1083.
[16] A. Dubey, W. Emfinger, A. Gokhale, P. Kumar, D. McDermet, T. Bapty, G. Karsai, Enabling strong isolation for distributed real-time applications in edge computing scenarios, IEEE Aerosp. Electron. Syst. Mag. 34 (7) (2019) 32–45.
[17] S. Bhatkar, D.C. DuVarney, R. Sekar, Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits, in: USENIX Security Symposium. Vol. 12, 2003, 291–301.
[18] R.N. Charette, This car runs on code, IEEE Spectr. 46 (3) (2009) 3.
[19] A. One, Smashing the stack for fun and profit (1996), 2007, See http://www.phrack.org/show.php.
[20] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, T. Holz, Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications, in: Security And Privacy (SP), 2015 IEEE Symposium On, IEEE, 2015, pp. 745–762.
[21] B. Potteiger, Z. Zhang, X. Koutsoukos, Integrated data space randomization and control reconfiguration for securing cyber-physical systems, in: Proceedings of The 6th Annual Symposium on Hot Topics in the Science of Security, ACM, 2019, p. 3.
[22] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, M. Caccamo, S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems, in: Proceedings Of The 2nd ACM International Conference On High Confidence Networked Systems, ACM, 2013, pp. 65–74.
[23] C. Gorgovan, A. D'antras, M. Luján, MAMBO: A low-overhead dynamic binary modification tool for ARM, ACM Trans. Archit. Code Optim. 13 (1) (2016) 14.
[24] Adubey14/arinc653emulator: This code base contains a linux emulator for the ARINC-653 operating system services, 2019, https://github.com/adubey14/arinc653emulator. (Accessed on 07/07/2019).

[25] C. Lattner, et al., The LLVM compiler infrastructure, 2010, http://llvm.org.
[26] M. Capelletti, Unlinker: an approach to identify original compilation units in stripped binaries, POLITesi (2017).
[27] A. Dinaburg, A. Ruef, Mcsema: Static translation of x86 instructions to llvm, in: ReCon 2014 Conference, Montreal, Canada, 2014.
[28] F. Markl, Case Study on LLVM as suitable intermediate language for binary analysis, Ret, 32.
[29] Y. Sui, J. Xue, SVF: Interprocedural static value-flow analysis in LLVM, in: Proceedings of The 25th International Conference on Compiler Construction, ACM, 2016, pp. 265–266.
[30] N. Mahadevan, A. Dubey, G. Karsai, Application of software health management techniques, in: Proceedings Of The 6th International Symposium On Software Engineering For Adaptive And Self-Managing Systems, ACM, 2011, pp. 1–10.
[31] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, V. Koltun, CARLA: AN open urban driving simulator, 2017, arXiv preprint arXiv:1711.03938.
[32] D. Isovic, 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, IEEE, 1999, pp. 60–67.
[33] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, S. Ioannidis, ASIST: architectural support for instruction set randomization, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013, 981–992.
[34] K. Lu, C. Song, B. Lee, S.P. Chung, T. Kim, 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, 2015, 280–291.
[35] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, A.-R. Sadeghi, Selfrando: Securing the tor browser against de-anonymization exploits, Proc. Priv. Enhanc. Technol. 2016 (4) (2016) 454–469.
[36] D. Seto, E. Ferreira, T.F. Marz, Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis), Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.
[37] B. Potteiger, Z. Zhang, X. Koutsoukos, Integrated instruction set randomization and control reconfiguration for securing cyber-physical systems, in: Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in The Science of Security, ACM, 2018, p. 5.
[38] B.D. Potteiger, A Moving Target Defense Approach Towards Security and Resilience in Cyber-Physical Systems, (Ph.D. thesis), 2019.
[39] M. Xia, P.J. Antsaklis, V. Gupta, Passivity indices and passivation of systems with application to systems with input/output delay, in: 53rd IEEE Conference on Decision and Control, 2014, 783–788.
[40] X. Koutsoukos, N. Kottenstette, J. Hall, E. Eyisi, H. Leblanc, J. Porter, J. Sztipanovits, A passivity approach for model-based compositional design of networked control systems, ACM Trans. Embed. Comput. Syst. 11 (4) (2012) 75.
[41] N. Kottenstette, J.F. Hall, X. Koutsoukos, J. Sztipanovits, P. Antsaklis, Design of networked control systems using passivity, IEEE Trans. Control Syst. Technol. 21 (3) (2013) 649–665.
[42] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S.A. Smolka, S.D. Stoller, E. Zadok, Software monitoring with controllable overhead, Int. J. Softw. Tool. Technol. Transf. 14 (3) (2012) 327–347.
[43] A. Bauer, M. Leucker, C. Schallhart, Model-based runtime analysis of distributed reactive systems, in: Software Engineering Conference, 2006. Australian, IEEE, 2006, p. 10.
[44] S. Tripakis, A combined on-line/off-line framework for black-box fault diagnosis, in: International Workshop on Runtime Verification, Springer, 2009, pp. 152–167.
[45] M. Wagner, P. Koopman, J. Bares, C. Ostrowski, Building safer UGVs with run-time safety invariants, in: National Defense Industrial Association Systems Engineering Conference, 2009.
[46] F.B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50.
[47] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Trans. Inf. Syst. Secur. 12 (3) (2009) 19.
[48] Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime? Int. J. Softw. Tool. Technol. Transf. 14 (3) (2012) 349–382.
[49] A. Zuepke, M. Bommert, D. Lohmann, AUTOBEST: A united AUTOSAR-OS and ARINC 653 kernel, in: 21st IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE, 2015, pp. 133–144.
[50] P.J. Prisaznuk, ARINC 653 Role in integrated modular avionics (IMA), in: 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, IEEE, 2008, pp. 1–E.
[51] N. Diniz, J. Rufino, ARINC 653 in space, in: DASIA 2005-Data Systems in Aerospace, vol. 602, 2005.
[52] M.-K. Yoon, S. Mohan, C.-Y. Chen, 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), IEEE, 2016, pp. 1–12.
[53] K. Krüger, G. Fohler, M. Volp, Improving security for time-triggered real-time systems against timing inference based attacks by schedule obfuscation, 2017.
[54] J.H. Jafarian, E. Al-Shaer, 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, 2012, 127–132.
[55] T.W. Yellman, Redundancy in designs, Risk Anal. Int. J 26 (1) (2006) 277–286.
[56] B. Blair, Strengthening checks on presidential nuclear launch authority, Arms Control Today 48 (1) (2018) 6–13.
[57] J. Barr, A. Narin, J. Varia, Building fault-tolerant applications on aws, Amaz. Web Serv. (2011) 1–15.
[58] A. Avizienis, The N-version approach to fault-tolerant software, IEEE Trans. Soft. Eng. (12) (1985) 1491–1501.
[59] C. Wang, H.-s. Kim, Y. Wu, V. Ying, Compiler-managed software-based redundant multi-threading for transient fault detection, in: Proceedings of The International Symposium on Code Generation and Optimization, IEEE Computer Society, 2007, pp. 244–258.
[60] L. Sha, Using simplicity to control complexity, IEEE Soft. 18 (4) (2001) 20–28.
[61] J. Yao, X. Liu, G. Zhu, L. Sha, NetSimplex: COntroller fault tolerance architecture in networked control systems, IEEE Trans. Indus. Infor. 9 (1) (2013) 346–356.
[62] X. Wang, N. Hovakimyan, L. Sha, L1simplex: fault-tolerant control of cyber-physical systems, in: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems, ACM, 2013, pp. 41–50.
[63] S. Bak, D.K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, L. Sha, The system-level simplex architecture for improved real-time embedded system safety, in: Real-Time And Embedded Technology And Applications Symposium, 2009. RTAS 2009. 15th IEEE, IEEE, 2009, pp. 99–107.
[64] M.-K. Yoon, B. Liu, N. Hovakimyan, L. Sha, Virtualdrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems, in: Proceedings of the 8th International Conference on Cyber-Physical Systems, ACM, 2017, pp. 143–154.