International Journal of Information Security
https://doi.org/10.1007/s10207-021-00568-1

REGULAR CONTRIBUTION

Data space randomization for securing cyber-physical systems

Bradley Potteiger[1] · **Feiyang Cai**[2] · **Zhenkai Zhang**[3] · **Xenofon Koutsoukos**[2]

Abstract
Non-control data attacks have become widely popular for circumventing authentication mechanisms in websites, servers, and personal computers. These attacks can be executed against cyber-physical systems (CPSs) in which not only authentication is an issue, but safety is at risk. Furthermore, any unauthorized change to safety-critical variables within the software may cause damage or even catastrophic consequences. Moving target defense techniques such as data space randomization (DSR) have become popular for protecting against memory corruption attacks such as non-control data attacks. However, current DSR implementations rely on source code transformations and do not stop critical variables from being overwritten, only that the new overwritten value will be vastly different than expected by the attacker. As such, these implementations are often ineffective for legacy CPS software in which only a binary is available. The problem addressed in this paper is how do we protect against non-control data attacks in legacy CPS software while ensuring that we can detect instances of variable integrity violations. We solve this problem by combining DSR at the binary level with variable comparison checks to ensure that we can detect and mitigate any attacker attempt to overwrite safety-critical variables. Our security approach is demonstrated utilizing an autonomous emergency braking system case study.

Keywords Moving target defenses · Data space randomization · Cyber-physical systems · Autonomous vehicles · Resiliency

1 Introduction

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

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

In existing DSR approaches, success is defined by the translation of adversary injected values into outlier data. However, if these data are still of a valid representation, it will not result in any program exceptions and even could

✉ Bradley Potteiger
bpottei1@umbc.edu

[1] Johns Hopkins Applied Physics Lab, Laurel, MD, USA

[2] Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA

[3] Department of Computer Science, Clemson University, Clemson, SC, USA

Published online: 06 November 2021

possibly satisfy existing detection constraints if the translated value happens to fall within a defined safe range. Additionally, existing approaches rely on source code transformations, but legacy software usually can only be accessed in a binary format. Due to the safety-critical nature of CPS and sheer volume of legacy code, these existing solutions fall short of being effective, allowing for unsafe repercussions to occur from cyber-attack attempts. As such, it is important to develop a methodology for performing DSR at the binary level, allowing for a dynamic randomization process at runtime, and providing re-randomization capabilities to further hinder adversary reconnaissance efforts and maintain availability.

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

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

In the past work, we developed a three-stage control architecture consisting of attack protection, detection, and recovery [35]. In this previous work, we utilized instruction set randomization (ISR) and address space randomization (ASR) to protect against code injection and code reuse attacks. Furthermore, we built in a reconfiguration scheme to ensure that when an attack occurs, we can detect and recover to a backup software version before unsafe CPS operation occurs. However, because these MTD techniques fail to protect against non-control data attacks, DSR becomes a critical defense. We create a software DSR implementation utilizing dynamic binary translation at runtime to randomize critical variables, and derandomize them for memory accesses. Additionally, detection capabilities are integrated to leverage a variable redundancy structure to identify instances of attacks. Finally, we integrate our approach with reconfiguration to transition execution to a backup safety controller during an attack attempt. The contributions of our paper are as follows:

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

Compared with our preliminary work with DSR [34], this paper provides significant extensions with regard to the static analysis and evaluation of our approach. With regard to static analysis, we provide an in-depth analysis of the various scenarios in which our approach will work, while providing an example illustration for the reader to follow the process. With regard to evaluation, we have upgraded our hardware-in-the-loop testbed to support more sophisticated autonomous vehicle scenarios with the addition of the CARLA open-source simulator. Finally, we have implemented caching and hashing techniques to reduce performance overhead and improve the scalability of our approach.

The rest of the paper is organized as follows: Section 2 introduces the background relating to DSR and the relevant attack surface of safety-critical CPS, Sect. 3 describes the DSR approach for protecting the integrity of program variables, Sect. 4 describes the runtime process for detecting non-control data attacks, Sect. 5 presents an implementation of our integrated DSR architecture, Sect. 6 utilizes an autonomous vehicle case study to demonstrate our DSR implementation, Sect. 7 presents related work, and Sect. 8 provides concluding remarks.

2 Problem formulation

With the introduction of CPS such as connected and autonomous vehicles, traditionally standalone systems are now becoming significantly reliant on software infrastructure and remote communication interfaces. Current automobiles

include over 100 million lines of code and 50–70 electronic control units (ECUs), similar to the level of a F35 fighter jet [11]. Due to the large investment required for redesigning a system from the ground up, automotive companies often attempt to build security on top of existing infrastructure, leaving a large amount of legacy code in the process. As such, attackers can leverage the large attack surface and lack of CAN bus authentication to gain entry to automotive networks, pivot to safety-critical ECU's, and disrupt the physical actuation of the vehicle.

One of the most significant vulnerabilities discovered from legacy code is the buffer overflow. Buffer overflows result from the absence of a limitation of the length of stored input in the C and C++ language, leading to the overwriting of adjacent memory locations on the stack. By overwriting adjacent memory locations, adversaries can inject instruction payloads directly (code injection [33]), redirect control flow to existing functions in the program (code reuse [37]), and overwrite adjacent program variables (non-control data attacks). Unlike code injection and code reuse attacks, it is more difficult to protect against and detect a non-control data attack due to the minimal change in program execution.

Through vulnerabilities like buffer overflows, attackers can manipulate non-control program variable data to alter program behavior without altering control flow. One common technique utilized to disrupt these types of attacks is DSR. DSR changes the internal or external representation of an application's data in such a way as to ensure that the semantic content is unmodified but unauthorized use, access, or modification is hindered [32]. For this to be accomplished, the format, syntax, encoding, and other properties of the data can be randomized.

As such, DSR acts similarly to ISR in using a key-based randomization and de-randomization process to encode variable data sensitive to attack. Each variable data object is randomized before it is written to memory and is derandomized after it is read from memory. Consistent with the ISR process, the randomization process can be accomplished by using an XOR operation with a randomization key [6,9]. Additionally, there is also the possibility of using other symmetric encryption algorithms such as those in the AES family to add further security to the application. DSR provides both the ability to use a common shared randomization key, but for enhanced security, each variable should be mapped to a unique randomization key.

2.1 Threat model

An exemplary vehicle system model includes six components: a sensor cluster, actuator cluster, driving controller, telematics control unit (TCU), remote function actuator (RFA), and RFID sensor. The sensor cluster provides critical data representing the current state of the vehicle such as

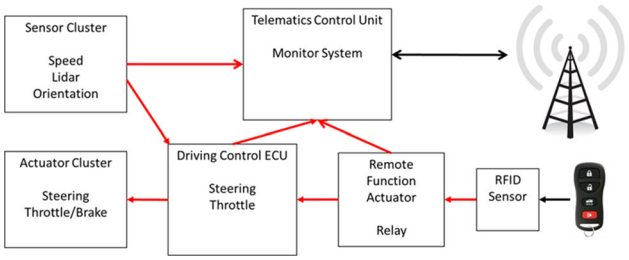

Fig. 1 Case study

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

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

The attack model for this paper focuses on a non-control data attack on a vehicle network. A non-control data attack is defined as an exploitation technique leveraging a buffer overflow vulnerability to overwrite adjacent variables to a non-bounds checked buffer, resulting in altered computation and control without the injection or redirection of code. Furthermore, these attacks are highly popular for remote exploitation, especially in the autonomous vehicle domain [30]. As such, an exemplar use case consists of the following attacker process flow. An adversary compromises a vehicle

TCU through the remote cellular interface and consequently pivots to hijack the RFA. With access to a direct communication channel with the driving controller, the adversary can craft a message payload to take advantage of the memory corruption vulnerability and alter control. At this point, an adversary can perform a non-control data attack where they can leverage the buffer overflow to overwrite adjacent safety-critical variables in the driving controller. As a result, at the next control iteration, the computation logic will utilize the modified variables, which leads to altered computation and actuation within the driving controller. By following this process, the goal of the adversary is to utilize the non-control data attack to cause the vehicle to perform unsafe behavior. At the same time, the goal of the defender is to prevent the non-control attack from succeeding while maintaining integrity of the driving controller software.

Four assumptions are made for our approach to be successful. First, it is assumed that the sensor and actuator clusters are fully secure. The driving controller ECU may contain a buffer overflow vulnerability utilized for control hijacking, while the TCU and RFA may contain vulnerabilities allowing for key fob message spoofing. Second, the attacker has knowledge of the relative address of a safety-critical variable relative to the start of the input buffer. Third, the attacker has knowledge of the underlying software architecture of the safety-critical controllers, allowing them to target the most impactful variables. Finally, the software cannot utilize dynamic memory allocation. These assumptions are not impractical given examples demonstrated in the literature [29] as well as popular CPS standards [3,23].

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

1. Any implemented software must maintain safe and reliable performance of the CPS. This includes minimizing the security architecture overhead and ensuring that all real time deadlines are met.
2. Implement reliable detection mechanisms for monitoring and flagging attack events.
3. Implement scalable defense mechanisms that can be extended to large programs.

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

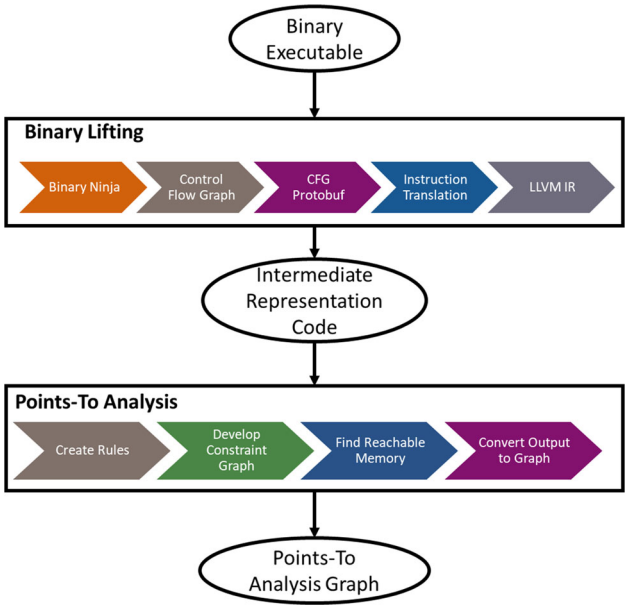

Fig. 2 DSR static analysis process

3 Static analysis for DSR

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

Our DSR approach is illustrated using an exemplary vehicle driving controller component (Fig. 3). This vehicle controller has several safety-critical variables that require randomization such as distance, speed, target speed, throttle, brake, orientation, target angle, steering, and key fob presence. Any alteration in any of these variables can result in direct safety violations in the autonomous vehicle driving behavior. As such, in the event of optimizing performance,

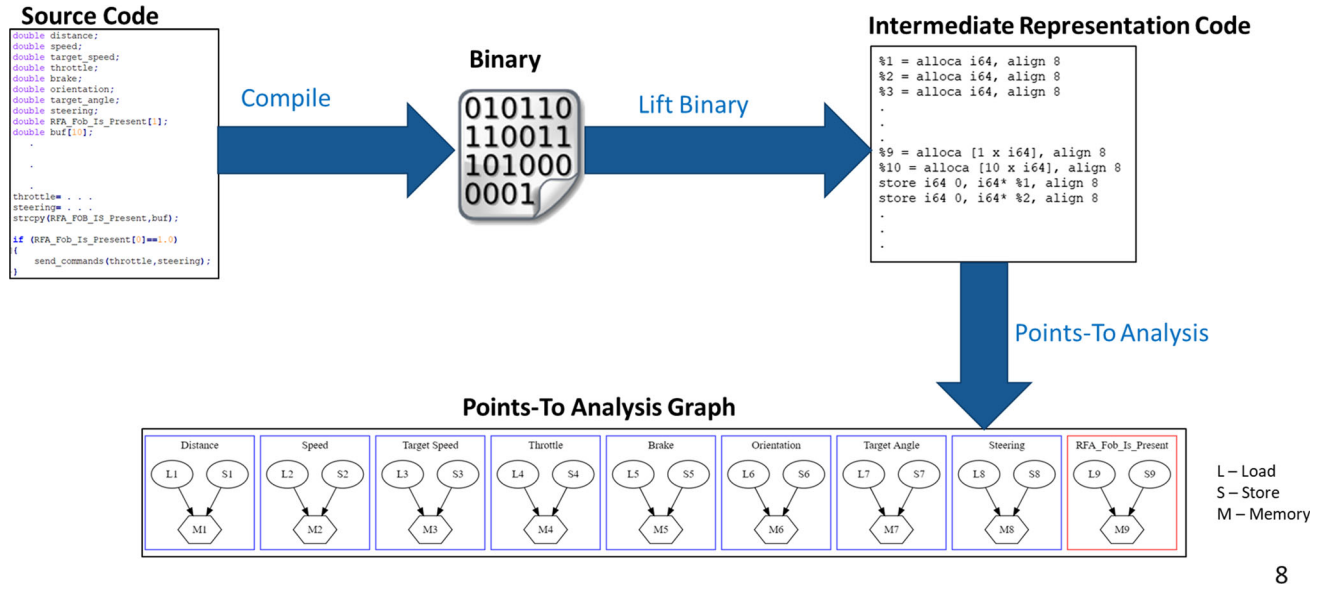

Fig. 3 DSR static analysis example

static analysis should prioritize the variables that reflect the most damage to the overall system and surround environment in the event of their compromise.

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

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

3.1 Binary lifting

In contrast to performing source-to-source transformations in C programs, it is difficult to manipulate and analyze variable instances in binary code. For the purpose of general analysis, it is optimal to convert the binary program into an intermediate representation (IR) format. The low-level virtual machine (LLVM) compiler includes an IR representation called LLVM bitcode, which we leverage for our analysis [27].

To lift a native binary to LLVM bitcode, we utilize an open-source tool called Mcsema [19,28]. Mcsema combines control flow recovery [10,20] and an instruction translation algorithm to directly convert instructions into LLVM bitcode.

3.2 Points-to analysis

The nodes extracted from the points-to analysis graph can either be a load or store instruction, or a memory region. As such, it is important when designing the data randomization process to understand the relationship between these different nodes. Due to being computationally undecidable, pointer analysis algorithms generally are approximations that provide varying degrees of precision and efficiency [36]. For our approach, we rely on a context-insensitive interprocedural points-to analysis implementation [2] utilizing an open-source static analysis tool called SVF [39].

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

3.3 Pointer scenarios

When analyzing a legacy software binary with points-to analysis, three possible scenarios can occur. The first scenario is that every encountered load or store instruction can only

point to one memory location. The second scenario is that it is possible for a load or store instruction to point to multiple memory locations, but any instruction pointing to a vulnerable buffer will not point to any adjacent memory location. The final scenario is an extension of scenario 2 in that instructions can point to multiple memory locations but instructions pointing to a vulnerable buffer will also point to adjacent memory locations.

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

Static analysis traditionally overapproximates a set of possible memory locations, meaning that if there is a non-bounded buffer, the analysis results will reflect that the instructions accessing that buffer have the potential to access all adjacent regions. However, due to bound analysis being included in the SVF analysis tool, the original analysis results will be pruned to reflect expected behavior, meaning that any instruction accessing the buffer will only be represented as accessing the respective buffer memory location [26].

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

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

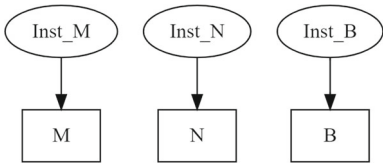

Fig. 4 Scenario 1

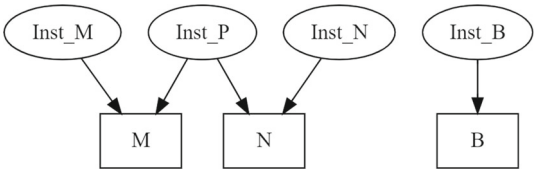

Fig. 5 Scenario 2

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

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

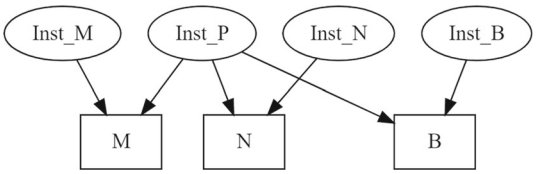

Fig. 6 Scenario 3

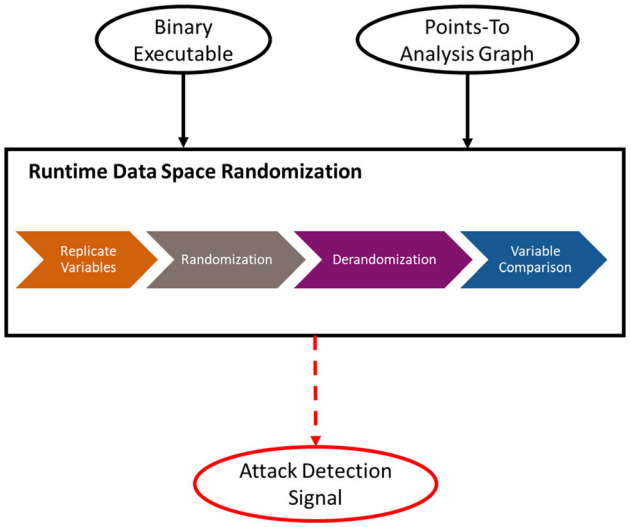

Fig. 7 Runtime process

4 Runtime randomization

The main vulnerability addressed by DSR is the overwriting and manipulation of adjacent variable data to input buffers [7]. The unique randomization and derandomization of individual variables ensure that if the attacker overwrites data, the semantic effect in the program will not be of the intended nature. For example, in the case of an adversary overwriting an adjacent target speed variable for an automobile, the desired goal could be increasing the value from 65 to 70 mph. In this case, the adversary will leverage a buffer overflow to insert the value 70 into the target speed variable memory location on the stack. However, in the case of DSR, the input buffer and target speed variable will have different randomization keys. Due to this fact, when the target speed is read from memory, it will first be derandomized with a different randomization key than what was utilized for writing, and the resulting value read will not be 70. An important note is that the resulting data values may still be of a valid format, meaning that an exception will not occur. However, the masking of variable values makes it easier for detection algorithms to determine the presence of a cyber-attack.

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

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

We leverage variable redundancy for the task of detecting non-control data attacks. For example, every time a variable is written to memory a duplicate variable instance will be written in the adjacent memory location. Two different randomization keys are assigned to these respective variable instances. During a variable load operation, the two variables will first be derandomized with their respective keys and compared for equivalency. In the event that they are equal, program execution can continue with variable access. However, in the event of the redundant variables being different the program is flagged for invalid behavior, execution is terminated without allowing variable access, and a backup controller resumes execution through a recovery process. By

utilizing this detection technique, it is not just enough for the adversary to correctly guess one variable randomization key, but must have knowledge of both keys to craft a successful payload.

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

5 Implementation

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

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

Dynamic binary translator (DBT) This component is responsible for providing a unique randomization backend for each spawned CPS controller in the architecture. In other words, the DBT is a virtual sandbox layer that serves as an intermediary between the executing binary and the processor. The DBT has the ability to intercept instructions as they are fetched and alter program semantics before execution by the processor. As such, a DSR methodology is supported by encoding variables before storage on the stack, as well as a derandomization stage before loading into registers. Each variable is supported to include a dynamically generated unique randomization key, as well as a duplicate comparison variable for detecting variable tampering. Additionally, the DBT is responsible for storing a variable key mapping table which incorporates uniquely generated keys for every variable in a program. We utilize an open-source dynamic binary instrumentation tool Mambo for our implementation [22].

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

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

5.1 Process flow

The basis behind the execution process of our security framework is a three-step approach: (1) static analysis, (2) binary load time, and (3) runtime. These steps are described below.

At design time, a significant amount of time needs to be dedicated to properly establish the CPS controller. This controller is located in secondary storage and is responsible for the control of the CPS based on sensor input and actuator output. Before loading the binary for execution, static analysis is performed to analyze the relationship between program variables with Mcsema and SVF [19,39]. At the conclusion of this step, a bipartite graphs of the relationships between load and store instructions and memory locations will be extracted from the PAG. This data structure is then iterated to identify any common instruction relationships between memory locations and will then create a set of memory address associations. For example, if two memory locations have an instruction in common, those memory locations will be combined into a set of memory addresses, which will share randomization key information.

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

MTD forms the backbone of our security architecture, incorporating DSR to protect against non-control data attacks

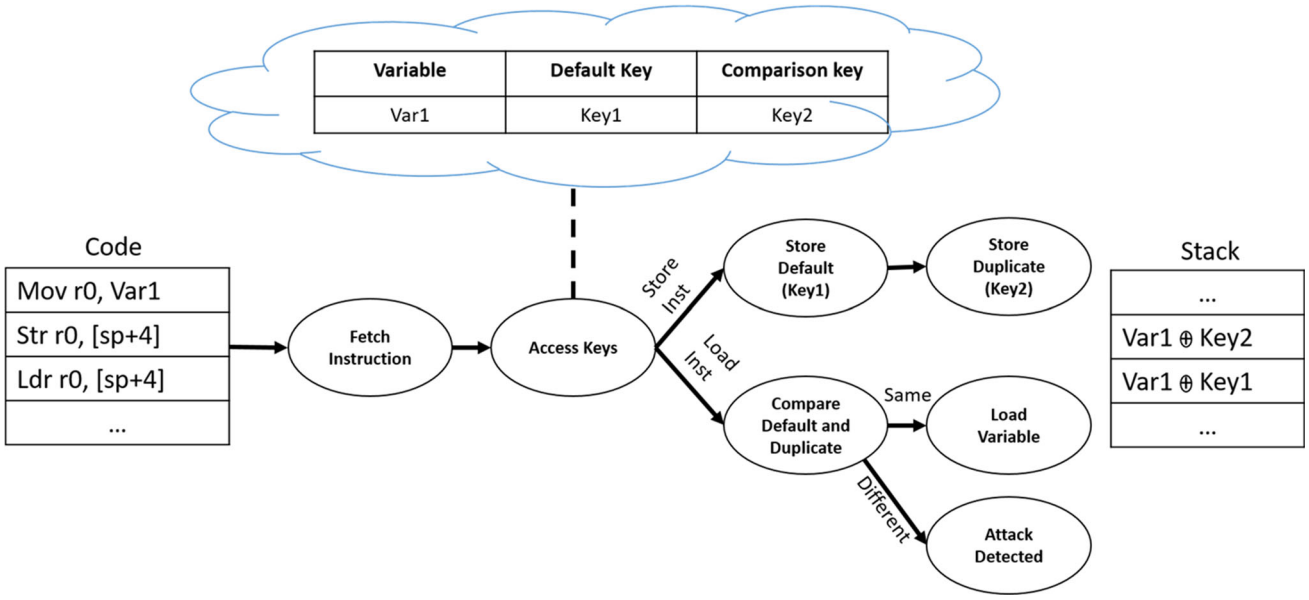

Fig. 8 Variable randomization process

as well as redundant controllers for reconfiguring during an attack. The goal is to decrease the probability of a successful cyber-attack by raising the level of effort needed by the adversary for obtaining accurate reconnaissance knowledge. Utilizing dynamic binary translators, which we implement utilizing the MAMBO DBT environment [22], local program variables can be randomized at store time by XORing the value with a dynamically generated key, as well as derandomized at variable load time by executing another XOR operation with the key. Since in between variable store and load time the variable will be in an encoded state, any effort by the adversary to alter the variable will result in a derandomized value wildly different than the attackers intended change. When looking at the randomization process by the MAMBO DBT enclosure, every time a variable store instruction is encountered two randomization keys will be retrieved from an internal variable map. These two keys will be associated with the original variable, as well as a redundant comparison variable. When storing to memory, the original variable will not only be stored in encoded form, but a redundant version of that variable will be stored in an adjacent location with a different randomization key. When a load instruction is encountered for the respective variable in the program, a check is first performed to determine whether the derandomized version of both the original and comparison variables is identical. Any change by the adversary will result in a failure in this comparison, resulting in the ability to detect the attack. Once an attack is detected, the the user has many options including terminating the process or spawning a backup process.

6 Evaluation

6.1 Experiment setup

To enhance the ability to evaluate cyber-attack impacts in deployment environments, a hardware-in-the-loop testbed was developed. Our testbed includes embedded hardware representing CPS software infrastructure, a simulation workstation representing the physical environment, and multiple network interfaces representing communication channels within the automobile environment. The setup of our testbed includes four components: 2 beaglebone black microcontrollers [15] representing the sensor and actuator processes in an automobile ECU cluster, a NVIDIA Jetson TX2 board [31] providing for computational power necessary for designing vehicle control algorithms, an i7 simulation desktop, and a real-time web-based results dashboard. Furthermore, two communication interfaces exist including 100-Mbps ethernet and a 1-Mbps CAN bus. The hardware architecture is illustrated in Fig. 9.

It is important to note that the decision to leverage the specific components (Ubuntu 18, NVIDIA Jetson, Beaglebone, etc.) came down to preference. This testbed can be scaled to several other setups with the following characteristics. First, the simulation desktop needs to be able to have enough processing and graphics power to run CARLA. At the moment, CARLA is compatible with the Windows 7 and 10,and

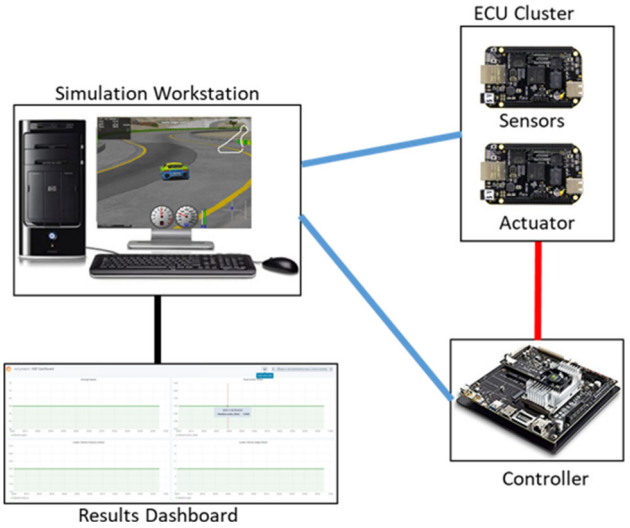

Fig. 9 Testbed hardware architecture

Ubuntu 16 and 18 distributions. However, as compatibility with this simulator increases, so will the number of compatible platforms. Second, the embedded boards (NVIDIA Jetson TX2, Beaglebone Black) can also be replaced with any other embedded computation board. With the absence of embedded boards, controller and actuation code can be hosted on the simulation desktop computer. In this case, a virtual CAN interface can be leveraged for the actuation communication.

6.1.1 Software architecture

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

Autonomous vehicle simulator The autonomous vehicle simulator utilized in our testbed is the CARLA autonomous vehicle simulator [21]. CARLA can be run on Windows and Linux, but for our setup we have the simulator running on Ubuntu 18.04. A socket-based communication is provided to access variables in the simulation, but we built a customized python API interface for easing variable access from external processes in the other distributed hardware in our testbed. The simulator can be customized to output sensor data such as lidar, speed, images, distance to objects, orientation, and GPS locations. Among the outputs, the user can change variables such as steering, acceleration, and braking. CARLA is the most sophisticated autonomous vehicle simulator to date and allows for us to develop more realistic experiments.

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

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

6.2 Case study

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

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

6.3 Attack scenario

As illustrated in Fig. 1, the follower vehicle is comprised of several components including a sensor and actuator ECU cluster, driving controller, telematics control unit (TCU), remote function actuator, and RFID sensor. There are two external interfaces including cellular communications from the TCU for remote monitoring services, and a RFID sync with the vehicle key fob. The driving controller constantly polls for the key fob signal to determine whether the engine should remain on. When the key fob is within a close distance, the vehicle will be able to drive, but as soon as the key fob is out-of-communication range the vehicle will turn off. Under normal operation, there is not a communication channel for the TCU to transmit input to the driving controller. However, since the TCU is connected remotely through a cellular interface, this component is the most at risk for being

compromised by the adversary. Even though the attacker cannot compromise the driving controller directly through the TCU, they can still utilize an intermediary step through the remote function actuator to inject malicious input into the driving controller. As such, the attack process consists of the following steps: (1) compromise TCU through cellular communications, (2) pivot to remote function actuator component, (3) transmit malicious input to driving controller, (4) overwrite distance values for AEBS controller. By utilizing the above process, the adversary can cause the follower vehicle to collide into the back of the leader vehicle, resulting in significant damage to both vehicles.

6.4 Results

6.4.1 Static analysis performance

For the purpose of the case study, a three-layer sequential neural network is utilized for the AEBS component of the driving controller. From an implementation standpoint, there are 1025 variables with a file size of approximately 220 kilobytes. Additionally, there are three shared libraries that we need to secure: Tensorflow Lite, libm, and ZeroMQ. This provides a good baseline for measuring the scalability of the different components in our static analysis pipeline.

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

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

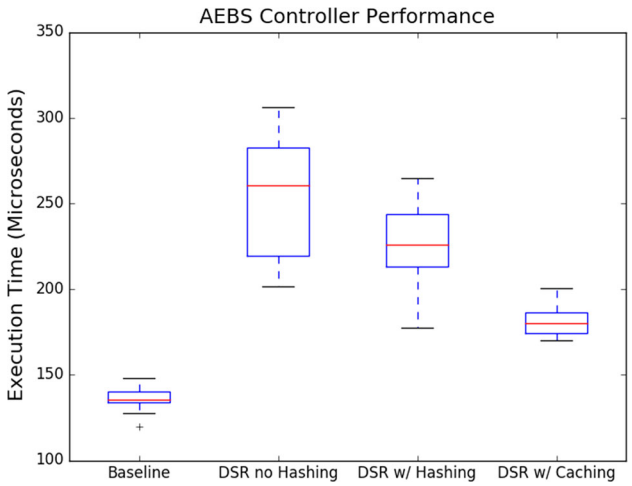

Fig. 10 Controller execution times

6.4.2 Experiment results

Due to the target sampling rate of 20 Hz, it is paramount to limit the overhead of our security architecture. As shown in Fig. 10, the overhead created with DSR enabled is pretty significant. To measure the overhead, we analyzed the execution times for 1000 iterations of our DSR approach with varying inputs. When looking at the AEBS controller execution times, overhead is approximately 83%, bringing the average execution time from approximately 135 microseconds for the baseline scenario to 247 microseconds with our DSR implementation enabled. Additionally, this overhead brings the worst-case execution time from 187 microseconds to 321 microseconds. Most of the overhead presented from our approach arises from accessing the randomization key lookup tables. To improve the overhead, we created a slight change within the lookup table to utilize hashing. We use a simple modulo approach with 100 slots to store randomization keys based on the memory location address that the instruction is accessing. With this approach, we were able to decrease our overhead from 85% to 61%, a 27% reduction. The average execution time with hashing is approximately 217 microseconds, while the worst-case execution time was 274 microseconds. Finally, to further improve the performance overhead we implement caching for the variable encryption keys. As such, instead of constantly accessing the lookup table, at the first instruction instance, we store the encryption key adjacent to the load/store instruction in program memory space. Therefore, every subsequent time this instruction is encountered will require only accessing the adjacent encryption key with an added instruction instead of utilizing program handlers to access the variable lookup table. For the first 100 iterations, this approach produces a performance overhead consistent with the hashing table approach, ranging around 60%. However, after the 100 iteration mark when

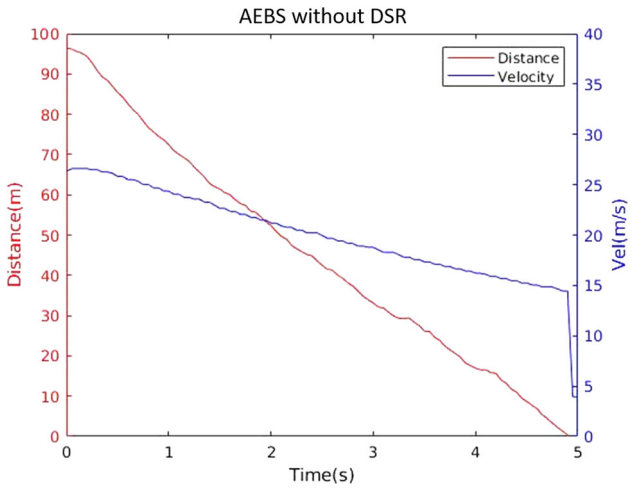

Fig. 11 Non-control data attack without DSR

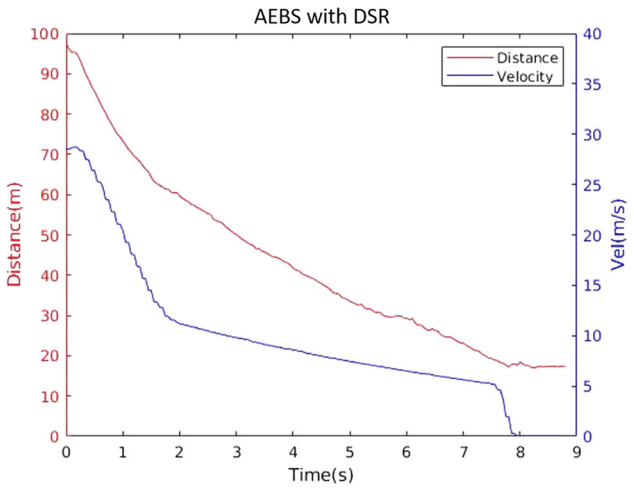

Fig. 12 Non-control data attack with DSR

a majority of the load and store instruction encryption keys were cached, we were able to further decrease our overhead to an average of approximately 34%, allowing for a more reasonable amount of overhead for real time applications.

During the case of a non-control data attack, the adversary is able to manipulate the AEBS controller operation by altering the perceived distance to the leader vehicle. With this adjustment, the new distance value is set at the max value of 100 meters, causing the follower vehicle to maintain acceleration. Furthermore, this continuance of acceleration combined with the rapid approach toward the leader vehicle will result in the crashing between the two vehicles as illustrated in Fig. 11. However, with DSR and variable integrity checking enabled, the attempt by the attacker to overwrite the distance variable will result in an incorrect variable comparison, consequently flagging the attack. At this point, a fail safe controller will take over execution and fully brake the vehicle. As a result, safety will be preserved and the follower vehicle will avoid colliding with the stopped leader vehicle as observed in Fig. 12.

7 Related work

In addition to code injection and code reuse attacks, non-control data attacks have been demonstrated to be realistic and devastating to the safety of CPS [13]. Data-oriented programming is especially concerning, allowing attackers to find a Turing complete approach to mount privilege escalation and information leakage attacks [25]. Through a quick analysis of 9 X86 programs, the authors were able to identify 7500 exploitable data gadgets [25]. Some basic defense techniques have been implemented in the literature to address these types of attacks including data watermarking [17], model-based anomaly detection monitors [18], and pointer taintedness detection [12]. However, the DSR literature seems to be the most promising with regard to preserving data integrity in the event of non-control data attacks [6].

Implementations of DSR started with a software toolkit called PointGuard [16]. PointGuard randomized the stored pointer addresses to prevent attackers from gaining reconnaissance knowledge about pointer data. In contrast, current DSR implementations not only randomize pointer addresses but encode the stored variable data [7]. The most popular implementation of DSR is integrated within the multicompiler, a customized LLVM compiler developed within the DARPA CFAR program [14] that can compile multiple unique binaries from the same source code [24]. However, current implementations are developed for source code, which poses a challenge when attempting to construct a dynamic security framework on binaries. To the best of our knowledge, our security framework is the first DSR implementation designed at the binary level. Some attacks against DSR listed in the literature include data leakage attacks, brute force and guessing attacks, and partial pointer overwrites [6]. With strategic derandomization and high randomization entropy, these types of attacks are deterred.

Redundancy has played a vital role for fault tolerance in safety-critical applications. Traditional past examples include airplanes [41], military protocols [8], and cloud servers [5]. 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 [4], and multithreaded-based implementa-

tions have demonstrated the ability to detect violations in application consensus [40]. 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.

8 Conclusion

In this paper, we have shown how DSR can be integrated with variable integrity checking techniques in autonomous vehicles for the purpose of ensuring secure and reliable operation. Due to the tightly coupled nature between the cyber and physical components in CPS, it is not just acceptable to maintain data integrity, but is necessary to guarantee a safe state of operation. Furthermore, non-control data attacks are a viable technique for altering physical behavior without the need for manipulating program control flow. Instead of overwriting the function return address on the stack, these attacks overwrite adjacent data variables to the input buffer with the goal of utilization in safety-critical operations. DSR can protect against non-control data attacks by changing the representation of variables, leaving adversary reconnaissance obsolete. As such, any manipulation of data variables will be vastly different compared to the intended goal. Furthermore, by including a duplicate stored variable with a different randomization key, a comparison can be performed at a variable load time to detect the presence of variable tampering. Our approach was tested with a hardware-in-the-loop testbed and an autonomous vehicle case study with an AEBS controller to illustrate CPS behavior on embedded hardware similar to deployment environments. By performing experimentation, we found that our framework produced positive security protection against non-control data attacks and limited physical behavior effects, while introducing reasonable performance overhead to the system. In the future, we plan to integrate our DSR security approach with ISR and ASR to include protections against code injection and code reuse attacks in addition to non-control data attacks. Further, these techniques are domain agnostic so can be applied to other domains such as the Power Grid [38] and Healthcare [1].

Funding This work was funded in part by the National Security Agency (H98230-18-D-0010), the National Science Foundation (CNS-1739328), and by the National Institute of Standards and Technology (70NANB17H266). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSA, NSF, or NIST.

Declarations

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Abbas, Z., Rehman, M.-U., Najam, S., Rizvi, S.D.: An efficient gray-level co-occurrence matrix (GLCM) based approach towards classification of skin lesion. In: 2019 Amity International Conference on Artificial Intelligence (AICAI), pp. 317–320. IEEE (2019)
2. Andersen, L.O.: Program analysis and specialization for the C programming language. Ph.D. thesis, University of Cophenhagen (1994)
3. Anderson, P.: Coding standards for high-confidence embedded systems. In: MILCOM 2008-2008 IEEE Military Communications Conference, pp. 1–7. IEEE (2008)
4. Avizienis, A.: The n-version approach to fault-tolerant software. IEEE Trans. Softw. Eng. **12**, 1491–1501 (1985)
5. Barr, P., Narin, A., Varia, J.: Building fault-tolerant applications on AWS. Amazon Web Services, pp. 1–15 (2011)
6. Bhatkar, S., Sekar, S.: Data space randomization. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2008)
7. Bhatkar, S., Sekar, S.: Data space randomization. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 1–22. Springer (2008)
8. Blair, B.: Strengthening checks on presidential nuclear launch authority. Arms Control Today **48**(1), 6–13 (2018)
9. Cadar, C., Akritidis, P., Costa, M., Martin, J.-P., Castro, M.: Data randomization. Technical report, technical report TR-2008-120, Microsoft Research, 2008. Cited on (2008)
10. Capelletti, M.: Unlinker: an approach to identify original compilation units in stripped binaries (2017)
11. Charette, R.N.: This car runs on code. IEEE Spectr. **46**(3), 3 (2009)
12. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., Iyer, R.K.: Defeating memory corruption attacks via pointer taintedness detection. In: 2005 International Conference on Dependable Systems and Networks (DSN'05), pp. 378–387. IEEE (2005)
13. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium, vol. 5 (2005)
14. Co, M., Davidson, J.W., Hiser, J.D., Knight, J.C., Nguyen-Tuong, J.C., Weimer, W., Burket, J., Frazier, G.L., Frazier, T.M., Dutertre B., et al.: Double helix and raven: a system for cyber fault tolerance and recovery. In: Proceedings of the 11th Annual Cyber and Information Security Research Conference, pp. 1–4 (2016)
15. Coley, G.: Beaglebone Black System Reference Manual. Texas Instruments, Dallas (2013)
16. Cowan, B., Beattie, S., Johansen, S., Wagle, P.: Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th Conference on USENIX Security Symposium, vol. 12, pp. 91–104 (2003)
17. Crandall, J.R., Chong, F.T.: Minos: control data attack prevention orthogonal to memory model. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 221–232. IEEE Computer Society (2004)
18. Demay, J.-C., Totel, J.-C., Tronel, F.: Sidan: a tool dedicated to software instrumentation for detecting attacks on non-control-data. In: 2009 Fourth International Conference on Risks and Security of Internet and Systems (CRiSIS 2009), pp. 51–58. IEEE (2009)
19. Dinaburg, A., Ruef, A.: Mcsema: static translation of x86 instructions to llvm. In: ReCon 2014 Conference, Montreal, Canada (2014)
20. Disassembler, I.P.: Debugger (2010)
21. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: Carla: An open urban driving simulator. arXiv preprint arXiv:1711.03938 (2017)

22. Gorgovan, C., D'antras, A., Luján, M.: MAMBO: a low-overhead dynamic binary modification tool for ARM. ACM Trans. Archit. Code Optim. TACO **13**(1), 14 (2016)
23. Hilderman, V., Baghi, T.: Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware). Avionics Communications (2007)
24. Homescu, A., Neisius, S., Larsen, S., Brunthaler, S., Franz, S.: Profile-guided automated software diversity. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–11. IEEE (2013)
25. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: on the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 969–986 (2016)
26. Lattner, C., Adve, C.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–86. IEEE (2004)
27. Lattner, C., et al.: The LLVM compiler infrastructure. http://llvm.org (2010)
28. Markl, C.: Case study on LLVM as suitable intermediate language for binary analysis. ret, 32:0
29. Miller, C., Valasek, C.: Remote exploitation of an unaltered passenger vehicle. Black Hat USA, 2015 (2015)
30. Miller, C., Valasek, C.: Securing self-driving cars (one company at a time). In: Presented at Black Hat (2018)
31. Naphade, M., Anastasiu, D.C., Sharma, A., Jagrlamudi, V., Jeon, H., Liu, H., Chang, M.-C., Lyu, S., Gao, S.: The NVIDIA AI city challenge. In: 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), pp. 1–6. IEEE (2017)
32. Okhravi, H., Hobson, T., Bigelow, D., Streilein, W.: Finding focus in the blur of moving-target techniques. IEEE Security and Privacy (2014)
33. One, A.: Smashing the stack for fun and profit (1996). See http://www.phrack.org/show.php (2007)
34. Potteiger, B., Zhang, Z., Koutsoukos, X.: 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, p. 3. ACM (2019)
35. Potteiger, B., Zhang, Z., Koutsoukos, X.: Integrated moving target defense and control reconfiguration for securing cyber-physical systems. Microprocess. Microsyst. **73**, 102954 (2020)
36. Ramalingam, G.: The undecidability of aliasing. ACM Trans. Program. Lang. Syst. TOPLAS **16**(5), 1467–1471 (1994)
37. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., Holz, A.-R.: Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in c++ applications. In 2015 IEEE Symposium on Security and Privacy (SP), pp. 745–762. IEEE (2015)
38. Shakir, M., Rehman, O.U., Abbas, Z., Masood, A., Shahid, W.: Evaluation of video quality in wireless multimedia sensor networks. Int. J. Electr. Comput. Eng. **6**(1), 223 (2016)
39. Sui, Y., Xue, Y.: SVF: interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 265–266. ACM (2016)
40. Wang, C., Kim, H.-S., Wu, Y., Ying, V.: Compiler-managed software-based redundant multi-threading for transient fault detection. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 244–258. IEEE Computer Society (2007)
41. Yellman, T.W.: Redundancy in designs. Risk Anal. Int. J. **26**(1), 277–286 (2006)