Integrated Data Space Randomization and Control Reconfiguration for Securing Cyber-Physical Systems

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

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

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

ABSTRACT

Non-control data attacks have become widely popular for circumventing authentication mechanisms in websites, servers, and personal computers. Moreover, in the context of Cyber-Physical Systems (CPS) attacks can be executed against not only authentication but also safety. With the tightly coupled nature between the cyber components and physical dynamics, any unauthorized change to safety-critical variables may cause damage or even catastrophic consequences. Moving target defense (MTD) techniques such as data space randomization (DSR) can be effective for protecting against various types of memory corruption attacks including non-control data attacks. However, in terms of CPS it is also critical to ensure the timely Cyber-Physical interactions after attacks thwarted by MTD. This paper addresses the problem of maintaining system stability and security properties of a CPS in the face of non-control data attacks by developing a DSR approach for randomizing binaries at runtime, creating a variable redundancy based detection algorithm for identifying variable integrity violations, and integrating a control reconfiguration architecture for maintaining safe and reliable operation. Our security framework is demonstrated utilizing an autonomous vehicle case study.

CCS CONCEPTS

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

KEYWORDS

Moving Target Defenses, Data Space Randomization, Cyber-Physical Systems, Resilient Architectures

ACM Reference Format:
Bradley Potteiger, Zhenkai Zhang, and Xenofon Koutsoukos. 2019. Integrated Data Space Randomization and Control Reconfiguration for Securing Cyber-Physical Systems. In *Hot Topics in the Science of Security Symposium (HotSoS), April 1–3, 2019, Nashville, TN, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3314058.3314064

1 INTRODUCTION

The design of safety-critical infrastructure is widely changing with the introduction of Cyber-Physical Systems (CPS). 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.

Memory corruption vulnerabilities like buffer overflows 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 this data is still of a valid format it will not result in any program exceptions and even could possibly satisfy existing detection constraints if the translated value happens to fall within a defined safe range. Additionally, existing DSR approaches rely on source code transformations, but legacy software usually can only be accessed in a binary format. 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.

In CPS, resiliency is a critical component to maintain system availability. Even though DSR can prevent non-control data manipulation, a failed attack can still lead to DoS behavior. In this case it is paramount to utilize proper fault tolerant recovery methods to ensure that system availability is maintained at all times.

The main problem that arises in this paper is how do we develop a DSR approach at the binary level to allow for providing a dynamic randomization capability. Further, we consider how to integrate DSR in combination with detection and reconfiguration to maintain system availability and real time behavior in the event of non-control data attacks. The main hypothesis of this paper is that by using DSR we can protect, and detect non-control data attacks while recovering control fast enough to maintain safe, and reliable system behavior.

In past work, we developed a three stage control architecture consisting of attack protection, detection, and recovery [21]. In this architecture we utilized Instruction Set Randomization (ISR), and Address Space Randomization (ASR) to protect against code injection and code reuse attacks. However, because these MTD techniques fail to protect against non-control data attacks, DSR becomes a critical element of providing protection. We leverage this work to create an attack resistent DSR based control architecture. We create a software DSR implementation utilizing dynamic binary translation at runtime to randomize critical variables, and derandomize them for memory accesses. Detection capabilities are integrated to leverage a variable redundancy structure to identify instances of attacks. Finally, a fault tolerant recovery algorithm is integrated for transitioning between redundant software implementations in the event of an attack. 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. Furthermore, we integrate DSR with a control reconfiguration architecture to defend against non-control data attacks while maintaining the resiliency and availability of safety-critical CPS
- 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, as well as the overhead presented to the system.

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, Section 3 intoduces the threat model for our paper, Section 4 describes the DSR approach for protecting the integrity of program variables, Section 5 describes the runtime process for detecting non-control data attacks, Section 6 describes the control reconfiguration process, Section 7 presents an implementation of our integrated DSR and control reconfiguration architecture, Section 8 utilizes an autonomous vehicle case study to demonstrate our DSR implementation, Section 9 presents related work, and Section 10 provides concluding remarks.

2 BACKGROUND

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 to 70 electronic control units (ECUs), similar to the level of a F35 fighter jet [9]. 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.

For example, one vulnerability discovered from legacy code is the buffer overflow. Buffer overflow's result from the absence of a limitation of the length of stored input in old versions of 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 [20]), redirect control flow to existing functions in the program (code reuse [25]), 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 [19]. For this to be accomplished the format, syntax, encoding, and other properties of the data are 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 [5, 7]. In these implementations the overhead is minimal with an average performance overhead of around 15% [30]. 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.

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

3 THREAT MODEL

For our threat model, we focus on legacy CPS controllers which may have memory corruption vulnerabilities such as buffer overflows. The attacker can leverage the vulnerabilities to overwrite and manipulate some adjacent safety critical variables by crafting and sending a malicious payload to the CPS controller. Consequently, the goal of the attacker is to manipulate the variable in such a way as to to bypass authentication, alter program output, or cause the CPS to enter an unsafe state.

In our threat model we make the following assumptions: 1) The CPS controller is the only vulnerable process in our system, containing a memory corruption vulnerability when processing incoming messages, 2) our security architecture, randomization keys, sensors, actuators, and underlying operating system are secure, 3) the communication channel can be compromised by the adversary, allowing for triggering vulnerabilities within the connected CPS controller, and 4) the CPS includes safety-critical actuation that can result in damaging consequences if compromised.

As a classical CPS, autonomous vehicles can be used to illustrate our threat model. An example autonomous vehicle model includes 5 main components: a sensor cluster, actuator cluster, driving controller, remote function actuator (RFA), and telematics control unit (TCU). Additionally, two communication interfaces exist: an external cellular channel for communication with the central maintenance station, and an internal safety-critical network for communication between the vehicle ECUs. The driving controller corresponds to the vulnerable CPS controller in our threat model, while the communication from the external cellular communication channel serves as the vulnerable remote communication interface where the attacker can gain entry into the system.

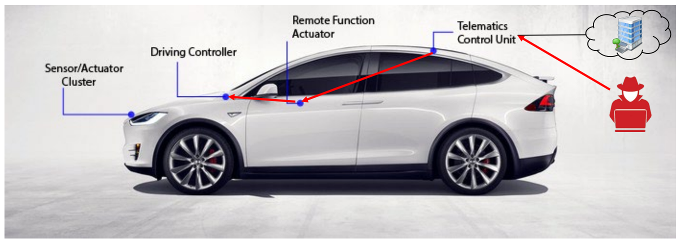

Figure 1: Autonomous Vehicle System Model

One possible attack vector consists of the following steps: 1) The attacker compromises the TCU through the remote wireless communication channel, 2) The adversary pivots to the RFA where they can manipulate the outgoing status messages to become a customized attack payload, 3) Malicious RFA status messages are sent to the driving controller which has an exploitable buffer overflow vulnerability allowing for the attacker to overwrite adjacent safety-critical variables in the program with a non-control data attack. In this case, the adversary overwrites the vehicle steering variable, causing the vehicle to lose control and drive off of the road. The vehicle system model and attacker message path are illustrated in Figure 1.

4 STATIC ANALYSIS FOR DSR

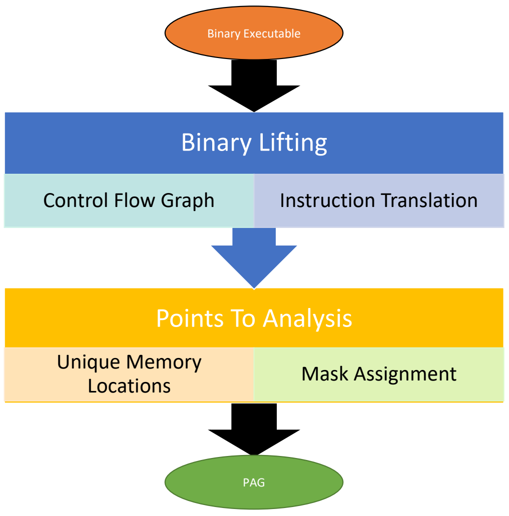

Figure 2: DSR Static Analysis Process

Our DSR approach is designed to transform program variables from 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, and points to analysis.

4.1 Binary Lifting

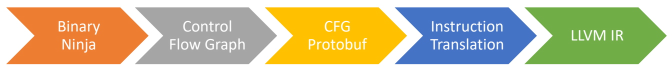

Figure 3: Binary Lifting Process

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 is the most widely utilized representation for this purpose [15]. Since many existing static analysis tools already use this IR, it makes sense to convert the binary program to this representation. This process is defined as binary lifting.

To convert native binary to LLVM bitcode two steps are necessary: control flow recovery, and instruction translation [17]. Control flow recovery includes analyzing the execution path of a program for the purpose of understanding the specifics of how a program functions. In the academic community it is standard to rely on an existing tool such as IDA Pro [13] or Binary Ninja [8] to compute the control flow graph due to the state of the art features and significant past work put in place. We continue this trend by relying on Binary Ninja for control flow recovery in our approach.

The instruction translation process uses a very simple approach, relying on a direct mapping between binary instructions and LLVM IR code. This decision intends to utilize the LLVM framework for program optimization which consequently reduces the required translation codebase. To translate code, we utilize an open source framework called Mcsema to perform the direct mapping between the binary and LLVM IR code [12].

4.2 Points To Analysis

In the context of a C based program, a variable can either be standalone data, or a pointer, As such, it is important when designing the data randomization process to understand the relationship between variables, and how many layers of unmasking are necessary. For example, in the simple case, a variable holding a data value would only need to be derandomized once to obtain the true value. However, in the case that the variable is actually a pointer to a variable at another address, not only does the pointer variable need to be unmasked to identify what address to access, but the actual value in the referenced memory needs to be derandomized as well.

Due to being computationally undecideable, pointer analysis algorithms generally are approximations that provide varying degrees of precision and efficiency [23]. There is also the tradeoff of performance with higher precision algorithms requiring a higher degree of time complexity. For example, algorithm factors such as flow-sensitivity, context-sensitivity, heap object modeling, aggregate object modeling, and aliasing increase the time complexity to exponential and high order polynomial. As such, algorithms with these factors are avoided in an effort to increase scalability.

The two most common pointer analysis algorithms are Steensgaard [28] and Andersen [2] which both provide flow and context insensitive inter-procedural points to analysis implementations. These algorithms both compute a points to set over named variables including local, global, and heap objects. In terms of comparison Steengaard's algorithm tends to be less precise, but performs better in terms of time complexity and scalability. However, since the goal of DSR is to randomize adjacent memory locations with different keys, it is very important to make sure that variable memory locations are segmented as much as possible. Therefore, in terms of our approach, precision becomes a priority making the Andersen algorithm a viable option for points to analysis.

After points to analysis is performed to determine variable relationships, an output is generated as a points to graph (PAG). With this PAG fed in as input, unique randomization keys are generated at load time for each respective variable object. Since data objects on the ARM processor are always 64 bits, the key size can be a consistent size compared to instances of other processors such as X86. With 64 bits of entropy, 2^{64} possible randomization keys can be generated which provides for a sufficient number of combinations for programs with a large amount of variables.

5 RUNTIME RANDOMIZATION

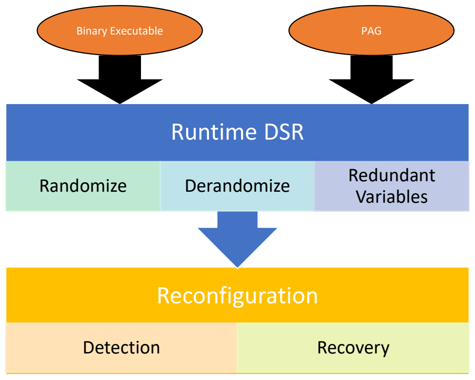

Figure 4: Runtime Process

The main vulnerability addressed by DSR is the overwriting and manipulation of adjacent variable data to input buffers [6]. The unique randomization and derandomization of individual variables ensures 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 mph 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 process for our runtime approach is illustrated in Figure 4. We focus on two steps: runtime DSR, and reconfiguration. These steps are described in more detail below.

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 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 relying on signals induced by illegal instruction execution and invalid memory address access, but a more active detection mechanism needs to be put in place.

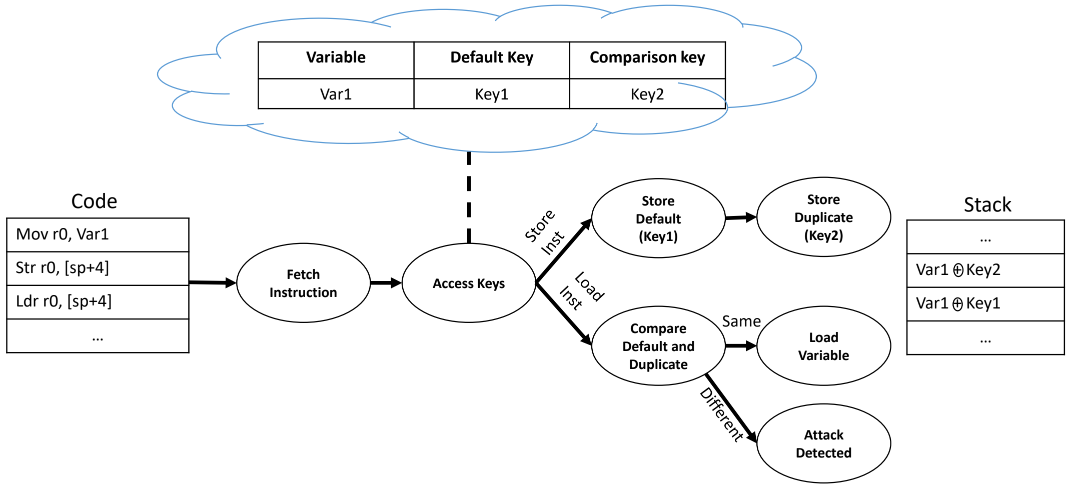

Figure 5: Variable Randomization Process

Variable redundancy can aid in the process of detecting non-control data attacks by utilizing different randomization keys for duplicate variable instances at store time, and performing a comparison of the derandomized values at variable load time. Any difference between the redundant variables during the comparison will indicate the presence of a variable manipulation. As such, to successfully bypass the comparison detection algorithm the adversary has to successfully guess both randomization keys correctly to ensure that the derandomized variables will be equivalent.

The primary defense mechanism utilized in our approach is the randomization and derandomization process of DSR which is illustrated in Figure 5. This means that when a variable 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. In other words, 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 load instruction is encountered, 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 additional XOR instruction is necessary to be inserted in the program before a store instruction, and after a load instruction. Additionally, encoding and decoding operations are only executed on register values, and not on the respective data in memory.

6 CONTROL RECONFIGURATION

Due to the existence of zero day exploits, security design must be converted from system hardening to a defense in depth approach. Security must be implemented at all levels of a system so if adversaries can get through outer layer protections, backup mechanisms are in place to protect the integrity of safety-critical operations. In CPS, it is not just enough to detect a cyber-attack but it is equally as critical to ensure availability. In a safety-critical application, a failed cyber-attack can still result in denial of service behavior, potentially leading to devastating physical consequences. Our approach is built upon a Simplex based recovery architecture, ensuring that in the event of a cyber-attack, there is always a path to resume valid program execution through a backup controller [21].

In our approach there are two spawned controllers at load time: a default high performance controller, and a backup safety controller. In general terms the default controller is more optimal for normal use but is not guaranteed to be completely secure, while the backup controller is less optimal from a performance standpoint but is guaranteed to be safe. This N version programming technique raises the difficulty for the adversary by making it necessary to not only find a vulnerability in one controller, but multiple different controller instances. Even though it is more beneficial from a security perspective to use different controller versions, our approach is not limited to a specific type of control algorithm, making it also possible to use the same controller version for both the default and backup controller.

At runtime the default controller will be spawned with the backup controller remaining in an idle state. Both controllers will contain a different set of randomization keys for their respective variables. When a cyber-attack is detected, execution in the default controller is terminated, the backup controller transfers to an executing state to serve as the new default controller, and a new controller is spawned to serve as the new backup controller. By using this process, it is guaranteed that a compromised controller will never execute, and the availability of safety-critical functionality will remain intact.

7 IMPLEMENTATION

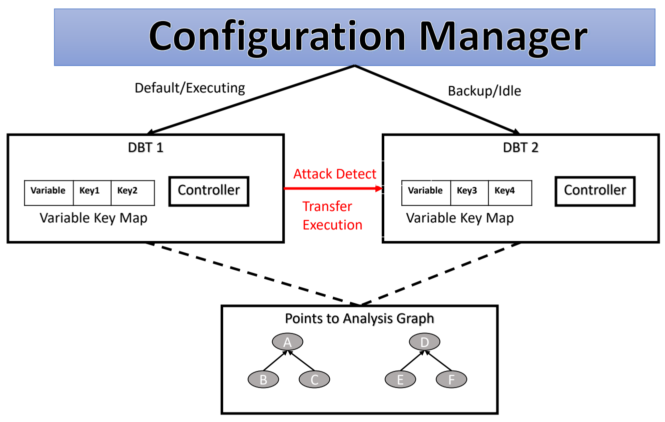

Figure 6: Architecture Design

Figure 6 presents an overview of our control architecture implementation which is based on a sequence of redundant controllers (a default and backup controller) and the transfer of execution during an attack. The key components in the architecture are the (1) Configuration Manager that oversees, customizes, and adjusts the operation of the various underlying controller components, (2) CPS Controller whichs control the physical plant, (3) DBT which uniquely customizes the runtime environment for each CPS controller, and (4) points to analysis graph (PAG) which describes the relationship between pointers and variables within a program.

Configuration Manager: This component is responsible for spawning the underlying CPS control applications, monitoring for the presence of cyber-attacks with a detection module, and transitioning execution between controllers with processes defined in the recovery module. The detection module incorporates a signal handler which is triggered when a program redundant variable comparison is not equivalent. Additionally, the recovery module is implemented to send a posix continue signal to the backup controller at the point of attack detection for the purpose of terminating the execution of the default controller and transfering execution to the backup controller. At this point, a new redundant controller will be spawned to serve as the new backup controller in the architecture.

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.

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 mediate any potential non-control data attack. As such, a DSR methodology is supported by encoding variables before storage on the stack, as well as a derandomizaton 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. The open source instrumentation tool Mambo is utilized to support the DBT implementation within our architecture [14].

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.

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.

7.1 Design Time

At design time, a significant amount of time needs to be dedicated to properly establishing the CPS controller. This controller is located in secondary storage and is responsible for the control of the CPS based on sensor input and actuator output. Before loading the binary for execution, static analysis is performed to analyze the relationship between program variables. The biggest challenge for variable randomization is to recognize the association between pointers and memory addresses. It is not just sufficient to randomize with respect to the memory storage location, but the pointer itself must be randomized. As such, during derandomization there are multiple steps of deencoding before access to the actual variable value is achieved. Static analysis in our security framework is achieved through 1) variable identification, and 2) points to analysis. For variable identification, iteration is performed over the Elf symbol table to access the full spectrum of variables within the target program. Afterwards, points-to-analysis is performed utilizing a combination of Mcsema [12], and SVF [29]. Mcsema is a binary lifter that utilizes a combination of control flow recovery and instruction translation to convert the program executable to an intermediate code representation that is compatible with current points to analysis algorithms in the LLVM compiler. Due to the vast amount of research put into solutions such as Binary Ninja [8], it makes sense to utilize these solutions for binary disassembly and control flow graph recovery versus developing a full disassembler from scratch. With the outputted control flow graph, Mcsema includes an open source instruction translation implementation to convert the underlying disassembly instructions into LLVM intermediate bitcode. After the intermediate bitcode is obtained, Anderson points to analysis is performed with the open source SVF library. SVF incorporates three steps consisting of graph, rules, and solver that takes a program control flow graph as input, formulates a set of variable constraints based on the Anderson methodology, and applies these constraints throughout the program utilizing a constraint solver in the LLVM compiler environment. After this stage, a GraphViz representation of the pointer and variable associations will be outputted in a dot file.

7.2 Load Time

At binary load time, each variable from the static analysis stage is stored in a mapping table within the respective 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.

7.3 Runtime

MTD forms the backbone of our security architecture, incorporating DSR to protect against non-control data attacks, 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 Translation Enclosures, which we implement utilizing the MAMBO DBT environment [14], 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, everytime 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 if the derandomized version of both the original and comparison variable are 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 configuration manager will disable operation of the current controller and transfer execution to a backup controller with a different set of variable randomization keys. This approach aims to reconfigure the controller fast enough to ensure stable operation of the CPS physical behavior.

8 EVALUATION

To sufficiently analyze CPS designs, simulation and emulation approaches are often utilized to link the software processes with the physical system behavior. We follow this approach by incorporating an autonomous vehicle case study with a hardware in the loop testbed setup. By integrating the testbed with a vehicle simulator, we can test various real world attack scenarios against our architecture, while also observing the effect on vehicle safety. Finally, we utilize several metrics including performance, vehicle position, and controller recovery time to evaluate the success of our approach.

8.1 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 [10] representing the sensor and actuator processes in an automobile ECU cluster, a NVIDIA Jetson TX2 board [1] providing for computational power necessary for designing vehicle control algorithms, an i7 simulation desktop, and a realtime 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 Figure 7.

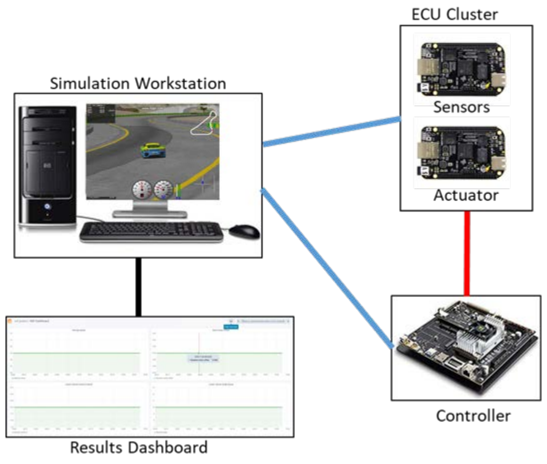

Figure 7: Testbed Hardware Architecture

8.1.1 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 TORCS Racing Simulator [32]. TORCS can be run on Windows, Linux, and Mac computers, but for our setup we have the simulator running on Ubuntu 16.04. A socket based communication is provided to access variables in the simulation, but we built a customized python API interface for easing variable access from external processes in the other distributed hardware in our testbed. The simulator can be customized to output sensor values such as lidar, speed, brake, gear, track position, distance from start position, vehicle heading, and position in the race. Among the outputs, the user can change variables such as steering, acceleration, braking, and gear value.

CPS Controller: The software for 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.

8.2 Case Study

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

The 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 steering, and throttle actuation, while receiving lidar, speed and orientation sensor readings as input. A PID controller is utilized to control driving behavior based on these inputs and outputs. However, our security architecture is generic meaning that it is not just limited to PID controllers, and any other controller software process can be utilized instead. The goal for the case study is to control the follower vehicle to maintain a consistent distance behind the leader vehicle, as well as staying as close as possible to the center of the road.

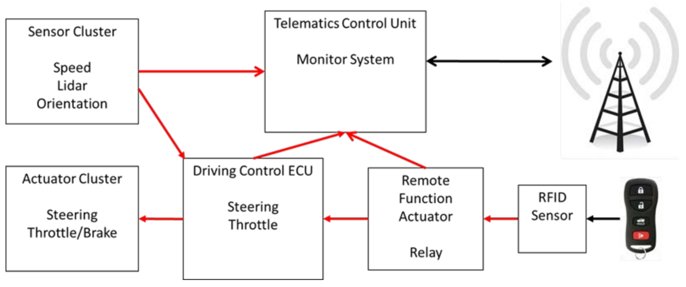

Figure 8: Case Study

8.3 Attack Scenario

As illustrated in Figure 8, 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 if 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 can't 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 target steering control value for PID controller. By utilizing the above process, the adversary can cause the follower vehicle to drive off the road, resulting in massive damage.

8.4 Results

8.4.1 *Static Analysis Performance.*

For the purpose of the case study a PID controller is utilized to control a vehicle. From an implementation standpoint, this is a fairly small program with a count of 15 local variables and file size of approximately 50 Kilobytes. This provides a baseline for measuring the performance of the different components in our static analysis pipeline. Furthermore, to explore scalability a 1 MB neural network controller developed from Tensorflow is evaluated with over 500 variables.

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 both the PID and neural network programs it appears that the execution time of the Mcsema custom section is pretty consistent averaging approximately 0.015 seconds for the PID controller and 0.018 seconds for the neural network program. 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 10 iterations of generating PAGs for the PID controller and neural network respectively. It was observed that the PID scenario produced an average execution time of approximately .12 seconds while the neural network controller produced an execution time of .34 seconds due to the significant increase of program variables to analyze. Even with this increased overhead the execution times are reasonable and are only necessary once during program runtime.

8.4.2 *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 Figure 9, the overhead created with DSR enabled

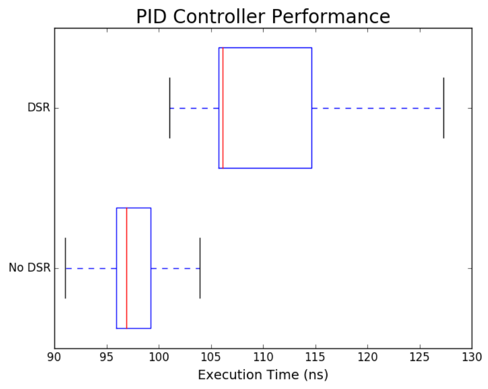

Figure 9: Controller Execution Times

is minimal. For example, when looking at the PID controller execution times, overhead is about 10%, bringing the average execution time from approximately 96 microseconds to 106 microseconds. Additionally, this overhead brings the worst case execution time from 105 microseconds to 128 microseconds. These results represent the lower bound of our architecture overhead. Even with a scaling factor of 10, this is still well under the 50 millisecond deadline defined in the design process. However, it is important to note that the PID controller is a relatively small sized program. With a large sized program, there is potential for the overhead to increase significantly.

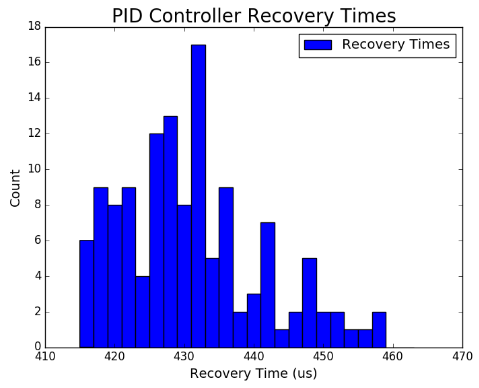

Figure 10: Recovery Times

During the case of a non-control data attack, the adversary is able to manipulate the PID controller operation by altering the target steering angle. With this adjustment, the new target steering value is set at the value of -1, causing the vehicle to make a hard left turn. Due to the fact that this value is constantly transmitted to the vehicle controller from the remote function actuator, the vehicle remains in a constant turning state and consequently performs donuts until crashing into the wall. However, in the scenario where DSR and reconfiguration is activated, the attack attempt will fail, and control will be transferred to a backup PID controller with a new randomization environment, decreasing the probability of a successful future attack. Furthermore, when looking at the damage of the follower vehicle behind the leader vehicle, the resultant attack disrupts the platoon behavior causing the following vehicle to be left behind in a crashed state. With reconfiguration enabled, the following vehicle continues to have reliable and safe operation. The vehicle driving behavior can be observed in Figure 11.

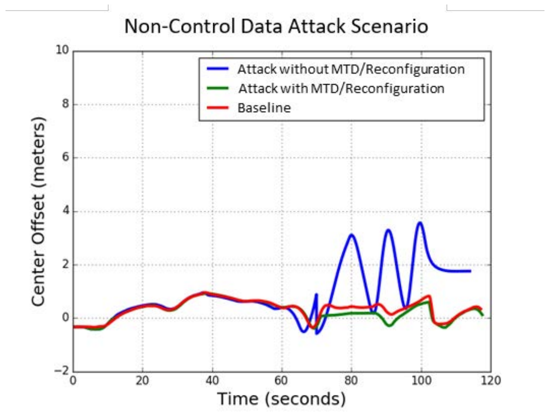

Figure 11: Road Center Offset Time Plot

9 RELATED WORK

Implementations of DSR started with a software toolkit called PointGuard [11]. 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 [6]. 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 [5]. With strategic derandomization and high randomization entropy these types of attacks are deterred.

With regards to recovery, there has been a wealth of work in the area of software fault tolerance. Several existing methodologies integrate N-Version programming to lower the probability of successive attacks by implementing different software versions with different structures, but similar semantics [3]. Additionally, checkpointing techniques such as recovery blocks have been utilized for rollback recovery implementations, allowing for controllers to maintain state through the reconfiguration process [16, 22, 24]. Simplex, which is the primary motivator of our security architecture, has been a widely utilized fault tolerant architecture, which consists of a complex controller, safety controller, and decision module which switches execution between the two based on specific events [27]. Several previous simplex based implementations include Secure System Simplex [18], Net Simplex [33], and L1 Simplex [31]. Furthermore, simplex architectures have been popular in safety-critical applications such as flight control systems [26], medical devises [4], and unmanned aerial vehicles [34].

10 CONCLUSION

In this paper we have shown how DSR can be integrated with control reconfiguration 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 variable load time to detect the presence of variable tampering. This method serves as the front line algorithm for attack detection in our framework. Finally, we utilized a redundant fault tolerant approach for integrating control reconfiguration into our framework. When an attack is detected, the configuration manager component successfully transfers execution to an idle backup controller, while a third controller is spawned to serve as the new backup controller. Our framework was tested with a hardware in the loop testbed and an autonomous vehicle case study 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 minimal performance overhead and recovery time to the system. In the future, we plan to integrate our security framework with ISR, and ASR to introduce protections against code injection and code reuse attacks in addition to non-control data attacks. Furthermore, we plan to test our DSR implementation against larger programs and benchmarks to identify how well it scales.

11 ACKNOWLEDGEMENTS

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

REFERENCES

[1] Jetson tx2 - elinux.org. http://elinux.org/Jetson_TX2. (Accessed on 11/15/2018).
[2] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.
[3] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.
[4] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 99–107. IEEE, 2009.
[5] S. Bhatkar and R. Sekar. Data space randomization. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2008.
[6] S. Bhatkar and R. Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.
[7] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical report, Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, 2008.
[8] M. CAPELLETTI. Unlinker: an approach to identify original compilation units in stripped binaries. 2017.
[9] R. N. Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.
[10] G. Coley. Beaglebone black system reference manual. *Texas Instruments, Dallas*, 2013.
[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.
[12] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.
[13] I. P. Disassembler. Debugger, 2010.
[14] C. Gorgovan, A. D'antras, and M. Luján. Mambo: a low-overhead dynamic binary modification tool for arm. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):14, 2016.
[15] C. Lattner et al. The llvm compiler infrastructure. *URL http://llvm. org*, 2010.
[16] M. R. Lyu. *Software fault tolerance.* John Wiley & Sons, Inc., 1995.
[17] F. Markl. Case study on llvm as suitable intermediate language for binary analysis. *ret*, 32:0.
[18] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pages 65–74. ACM, 2013.
[19] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving-target techniques. *IEEE Security and Privacy*, 2014.
[20] A. One. Smashing the stack for fun and profit (1996). *See http://www. phrack. org/show. php*, 2007.
[21] B. Potteiger, Z. Zhang, and X. Koutsoukos. Integrated instruction set randomization and control reconfiguration for securing cyber-physical systems. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, page 5. ACM, 2018.
[22] L. L. Pullum. *Software fault tolerance techniques and implementation.* Artech House, 2001.
[23] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.
[24] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, (2):220–232, 1975.
[25] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.
[26] D. Seto, E. Ferreira, and T. F. Marz. Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis). Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.
[27] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.
[28] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.
[29] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.
[30] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.
[31] X. Wang, N. Hovakimyan, and L. Sha. L1simplex: fault-tolerant control of cyber-physical systems. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, pages 41–50. ACM, 2013.
[32] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4, 2000.
[33] J. Yao, X. Liu, G. Zhu, and L. Sha. Netsimplex: Controller fault tolerance architecture in networked control systems. *IEEE Transactions on Industrial Informatics*, 9(1):346–356, 2013.
[34] M.-K. Yoon, B. Liu, N. Hovakimyan, and L. Sha. Virtualdrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pages 143–154. ACM, 2017.