CDP - ADAPTIVE SUPERVISORY CONTROL AND DATA ACQUISITION (SCADA) TECHNOLOGY FOR INFRASTRUCTURE PROTECTION

Final Report
Agreement No. 10OE0000511

Period of Performance:
09/15/2010 - 05/31/2012
Final Report, May 2012

Principal Investigator (Technical Contact)
Dr. Marco Carvalho
Research Scientist
Florida Institute for
Human and Machine Cognition
Phone: (850) 202-4446
mcarvalho@ihmc.us

Co-Principal Investigator
Dr. Richard Ford
Harris Professor of Computer Sciences
Harris Institute for Assured Information
Florida Institute of Technology
Phone: (321) 674-7473
rford@fit.edu

Administrative Contact
Diana Thacker
Institute for Human & Machine Cognition
40 S. Alcaniz St.
Pensacola, FL 32502
Tel: (850) 202-4473
Fax: (850) 202-4440
Email: dthacker@ihmc.us

Institute for Human and Machine Cognition
40 South Alcaniz St. - Pensacola, FL   32502
http://www.ihmc.us

Table of Contents

List of Figures

1. Project Overview

Supervisory Control and Data Acquisition (SCADA) Systems are a type of Industrial Control System characterized by the centralized (or hierarchical) monitoring and control of geographically dispersed assets. SCADA systems combine acquisition and network components to provide data gathering, transmission, and visualization for centralized monitoring and control. However these integrated capabilities, especially when built over legacy systems and protocols, generally result in vulnerabilities that can be exploited by attackers, with potentially disastrous consequences. The security risks associated with conventional SCADA systems are well recognized today, and there is a clear need for new, game-changing defense technologies in this area.

There are basically two ways to think about security for SCADA systems and control infrastructures. First, we may consider traditional defense approaches and create firewalls, anti-malware systems, and automated patching and configuration management products that are specialized to the control system needs. While helpful to some extent, conventional security mechanisms are just not sufficient, by themselves, as a solution to the problem. If circumvented (which is often the case) they become fully ineffective against an attacker. Unfortunately, this approach to network defense tends to a false sense of security, and potentially enables catastrophic attacks against the infrastructure.

Alternatively, we can also think about security of SCADA systems by recognizing that the actual goal of the attacker is to damage, exploit or in some way seize control of the underlying process that the system is managing. Attacking the nodes that make up a process control system is only a means to an end. Thus, an effective approach to survivable SCADA infrastructures must focus on the defense of the controlled process as a primary goal, and the defense of monitoring/control infrastructure simply as a means to an end. This allows for the prioritization of specific functions.

Our research project proposal was to investigate new approaches for secure and survivable SCADA systems. In particular, we were interested in the resilience and adaptability of large-scale mission-critical monitoring and control infrastructures.

2. Proposed Approach

As described in our statement of work, our research proposal was divided in two main tasks, with specific subtasks. The first task was centered on the design and investigation of algorithms for survivable SCADA systems and a prototype framework demonstration. The second task was centered on the characterization and demonstration of the proposed approach in illustrative scenarios (simulated or emulated). Each of these tasks and related sub-tasks are described below.

2.1 Task 1.0 Investigation and Design of Methods and Algorithms for SCADA Defense

Create new methods for defending SCADA systems that are mission centric. This task is subdivided into three smaller but interlinked subtasks. Collectively, these subtasks will result in a system design that can withstand attack by a determined adversary, and that is measurably more resilient than existing technologies.

2.1.1 Subtask 1.1 Development of conceptual framework for protecting SCADA Systems

In order to create a defensive framework, the Recipient will research countermeasures at different levels of the protected SCADA systems (for example, at the host, at the network, and at the protocol). Then, loosely break these down into high-level (collective) defenses, protocol-based defenses, and endpoint defense. At the endpoint, the Recipient will explore the application of a modified Harvard Architecture running on a virtualized chipset as an effective method of creating foundation for SCADA systems.

2.1.2 Subtask 1.2 Development of distributed algorithms

Given that a SCADA system is physically distributed, the Recipient will develop algorithms that allow these scattered assets to work together to share security-related information. This process is complicated by topology, computational resource constraints and potentially unreliable connectivity under attack. Leveraging previous research efforts targeted to tactical environments, the Recipient has investigated different approaches for distributed monitoring and collaborative learning. The Recipient plans to leverage some of those capabilities to enable defense algorithms with the necessary features to operate in distributed (and possibly dynamic) environments.

2.1.3 Subtask 1.3 Research potential coupling mechanisms between levels

One of the benefits of the proposed approach is that it will enable information sharing between different levels of the system. Cross-layer defense infrastructures can benefit from feedback collected at different levels to better estimate the causes of a perceived anomaly, and to better predict the effects of a potential course of action. This capability will allow the system to attain a security stance that is more effective than the sum of its parts. This approach is inspired by biological systems, where contributions at many different levels add together to produce a result that is robust under perturbation. Under this task, the Recipient will explore different coupling mechanisms that enable such functionality.

2.2 Task 2.0 – Characterization and Experimental Evaluation of Methods

Once the algorithms have been developed and are capable of improving SCADA system resilience to attacks and perturbation, the Recipient will characterize the system in order to demonstrate the efficacy of the approach. This is broken down into the subtasks shown below.

2.2.1 Subtask 2.1 Metrics Development

In order to appropriately measure the improvement in system resilience, the Recipient intends to develop metrics that can be used to meaningfully measure the security and resilience of the protected SCADA system. Metrics for software security in general are lacking; in this task the Recipient will attempt to extend the work that is extant in security metrics to measures that correlate with process stability.

2.2.2 Subtask 2.2 Investigations of System Properties

Understanding the properties of the design is critical to evaluation and improvement. Leveraging the metrics produced from Subtask 2.1, the Recipient will use mathematical methods, along with emulation and direct experimentation to understand more fully the properties of the proposed approach.

3. A Multi-Layer Defense Framework for Resilient Systems

In the context of this project, mission survivability refers to the ability of a system to maintain mission execution to its successful completion, and to remain mission capable despite localized system failures or attacks. For SCADA systems, a mission represents the monitoring and control of the underlying plant or system.

While there is an expected correlation between the survivability of the individual components which make up the physical system and the survivability of the mission, there is a conceptual difference between the two. The protection of the infrastructure implies the defense of each of the infrastructure assets, regardless of their role with respect to a particular task. The protection of the mission, on the other hand, seeks to ensure that its execution is carried out completely and successfully, with no concerns on the state of the underlying computational and communications infrastructure.

It is important to note, however, that an underlying system degradation or compromise will eventually lead to mission failure - or to the inability to support a mission. Thus mission survivability includes a contextualized survivability of the system. From this perspective, the notion of mission survivability (and resilience, as it will be later discussed) requires two core capabilities in face of an attack or local failure:

- a short term adaptive response, reallocating services and resources to ensure mission continuity while the attacked area is being addressed.
- a parallel defense mechanism that seeks to identify and isolate the attack, while learning attack patterns to estimate the level of vulnerability of other nodes.

Conventional strategies for survivable systems tend to rely on redundancy and service replication, often from an application perspective with little (if any) feedback from the dynamics of the network, ongoing attacks, or changes in system threats.

In this class of strategies, the assumption of equal probability of failure between nodes leads to a likelihood of survivability of a critical component that is directly proportional to the number of replicas. However, in most scenarios the likelihood of failure and vulnerability is far from uniform across the system and, in fact, may change as a function of the mission being executed.

Leveraging from our previous research in biologically inspired security infrastructure for tactical operational environments, we have defined a three-layer defense architecture that enables the survivability and protection of mission-critical distributed systems (Figure 1).

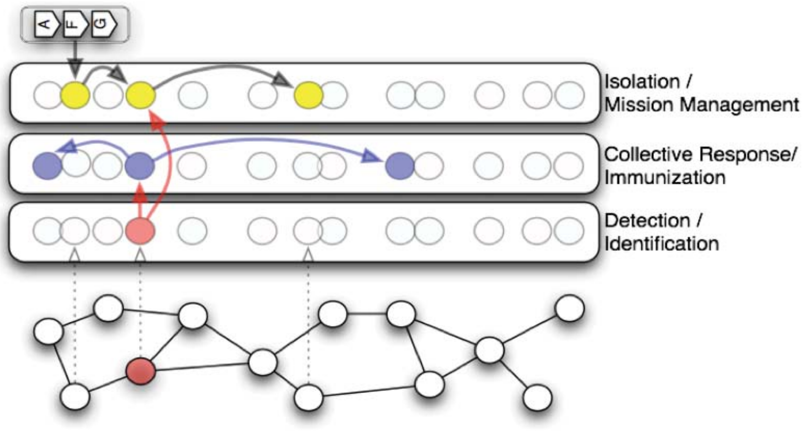

A
F
G
Isolation / Mission Management
Collective Response/ Immunization
Detection / Identification

Figure 1. Proposed Multi-Layer Defense Infrastructure for Survivable SCADA Systems

The lower layer provides the mechanisms for damage, or attack detection. Once detected, a localized damaged will trigger a parallel re-alignment of system resources to maintain mission execution while isolating the problem (upper layers), and an identification of the threat to initiate a recovery mechanism, as well as the detection and proactive mitigation of similar or related vulnerabilities (lower and middle layers). The re-alignment of system resources enables, for example, a new configuration of services required to satisfy a specific task, or mission.

3.1 The Concept of Organic Resilience

The proposed multi-layer approach to system resilience builds on peer-to-peer discovery and orchestration strategies for mission management and threat mitigation. It allows for the dynamic instantiation of services for redundancy and diversity, in response to localized attacks or failures.

The approach was first introduced in the context of resilient tactical infrastructures [1]. It is biologically inspired in the sense that we combine insights from developmental biology, diversity and immunology, including inflammatory and immunization systems. In our formulation these biological traits are desirable capabilities that can be implemented in multiple ways by leveraging services and features enabled by cloud computing and our own support services.

3.1.1 Detection and Identification Layer

One of the assumptions of our approach is that individual services are capable of monitoring their own sensors and performance to detect local damage. In practice, damage detection may be implemented in multiple ways. In the context of mission continuity, damage is directly related to the inability of a service to execute its tasks, or a significant degradation in task execution performance (below acceptable QoS requirements). From that perspective, there is no distinction between damage caused by localized failures or malicious acts. The effects of both events will be similar, as well as the way in which the system will respond.

Other approaches for damage detection have also included statistical and biologically inspired techniques based on Danger Theory [2], and Artificial Immune Systems [3][4] amongst others. In most cases damage is based on negative signature matching or anomaly detections associated with misbehavior or performance degradations. Upon damage detection the system will immediately notify the upper layers (for resource management and response/immunization), while in parallel trying to identify correlated features that could be linked (maybe causally) to the event. Previous research efforts have been proposed for that, including the application of Hidden Markov Models [5][6], decision trees [4][7], and others.

3.1.2 Threat Isolation and Mission Management

Automatic resource and service re-allocation in response to localized failures is common practice in Grid environments, and has also been previously proposed for enterprise [8] and tactical [9] environments. However, in general, a change in allocation strategy happens only when degradation (or failure) has taken place and the impact on the mission has been noted; there is generally no predictive re-allocation based on increased risk of an attack or failure, learned at runtime from novel attacks.

Our approach leverages and extends such dynamic allocation strategies to enable the proactive task reallocation, based on online risk estimations. For our current proof-of-concept mission management layer implementation, we have adopted a greedy distributed coordination algorithm using a generalized cost metric per node for resource management. When a workflow is received, each node makes a local decision about task execution based on current local cost estimations. If local costs become less attractive than neighbor's estimated costs then the workflow is forwarded to the node with the lowest estimated cost. Cost information is shared between nodes involved in a joint mission as part of workflow exchange messages.

Attacks and failures may be detected indirectly, through their effects on the mission (see 3.1). To simplify our model, we currently consider the degradation of a task as causing direct impact in the mission performance. There are, however, related research efforts on mission mapping [10] [11] [12] that can provide a better assessment of the impact of localized failures to the overall mission.

In general, the approach for detection may rely on a number of sources that include performance monitoring, anomaly detection, or resource utilization monitoring. These are all metrics that may be used to detect violations in resource utilization policies, or deviations from pre-defined (or learned, in the case of anomaly) QoS requirements for task execution.

Dynamic resource management for mission continuity focuses on isolating the area (i.e. node, or services) associated with the damage to minimize the impact on mission execution. The re-allocation of resources and re-organization of tasks is coordinated through distributed, self-organizing algorithms and may take place at different scales – that is, from very localized modification involving a single service that has reported damage, to larger scale changes involving multiple services. An analog to this approach can be found in developmental biology, where cells (and other structures at different hierarchical levels) signal each other to induce a differentiation that will enable a needed capability. In our approach, mission-aware services will perceive the lack of damaged capability and will signal other services (as part of a distributed orchestration mechanism) to engage the new capabilities.

3.1.3 Threat Mitigation and Immunization

The response and immunization mechanisms are responsible for both a short-term response to the reported damage and a longer-term mitigation strategy to future attacks of the same type. The intuitive response to a damaged component that can be replaced by alternative services in the environment is to immediately terminate the affected service. However, depending on the type of attack, the response and immunization layer may benefit from maintaining a potentially compromised node in operation.

The goal is to identify the potential causes of the effects perceived as damage, and possibly correlate those events with the configuration of the node. This rudimentary approach to vulnerability estimation is useful in providing a hint to other services in the system that may be equally vulnerable to the same types of attacks. In our proposed infrastructure, the response and immunization mechanism work together to allow some time for the system to build such correlations before shutting down the node as a response to the damage. In order to do that without affecting the mission, a duplicate of workflows (which have been re-allocated to alternative nodes in response to the damage) is still sent to the damage node for processing, but it is also tagged to be ignored by subsequent processing services. This allows for the damaged component to remain “active” for the characterization and immunization tasks

3.2 Additional Multi-Layer Defense Capabilities for SCADA Systems

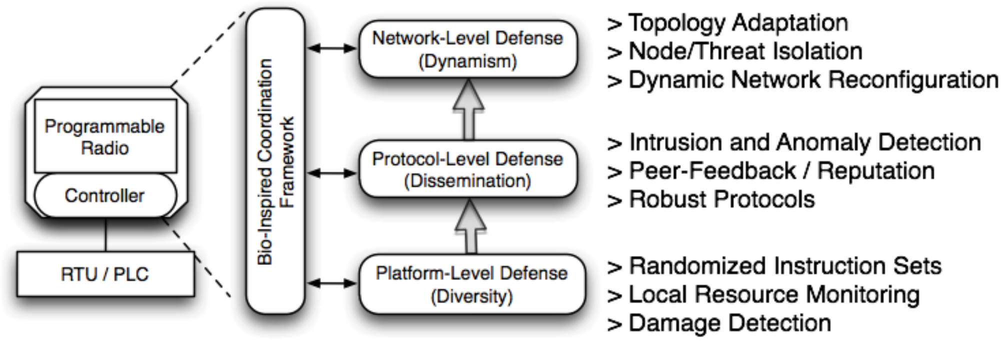

Programmable Radio
Controller
RTU / PLC
Bio-Inspired Coordination Framework
Network-Level Defense (Dynamism)
Protocol-Level Defense (Dissemination)
Platform-Level Defense (Diversity)
> Topology Adaptation
> Node/Threat Isolation
> Dynamic Network Reconfiguration
> Intrusion and Anomaly Detection
> Peer-Feedback / Reputation
> Robust Protocols
> Randomized Instruction Sets
> Local Resource Monitoring
> Damage Detection

Figure 2. Initial conceptual design for the proposed architecture, enabling the multi-layer resilience framework.

We started by identifying the architectural requirements to implement the proposed multi-layer defense, and also some of the core enabling capabilities for our approach (Figure 2).

One key requirement identified for the proposed architecture is the capacity to detect (and possibly identify) local damage to the system. Damage detection enables the system to benefit from the capabilities provided by the second and third layers.

As a first approach, we are focusing on damage detection from a) a host perspective, where new resilient architectures may be necessary to address the requirements, and b) the network perspective (i.e. network attacks, detected from specialized anomaly, and intrusion detection systems).

As previously reported, the coordination defense framework was designed to include three layers, as illustrated in Figure 3.

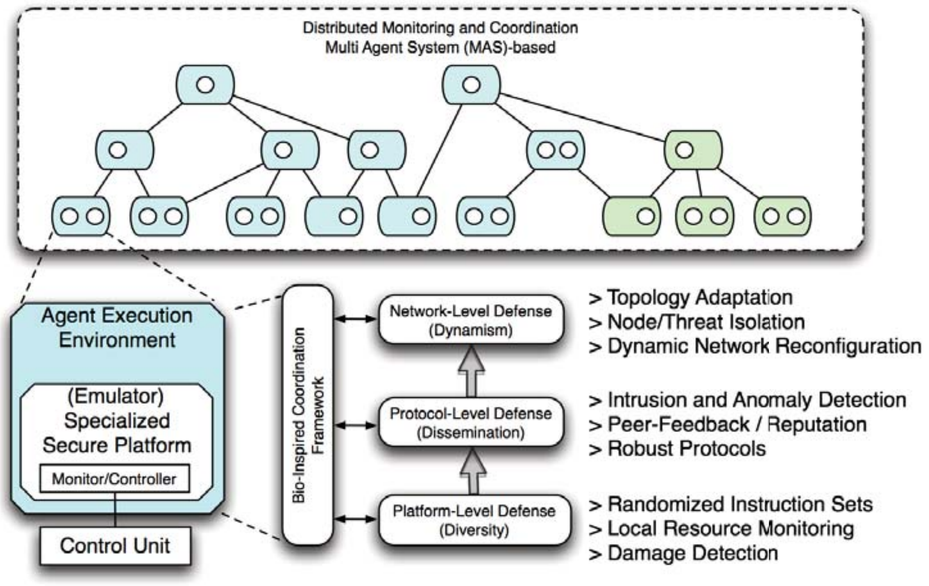

Distributed Monitoring and Coordination
Multi Agent System (MAS)-based
Agent Execution Environment
(Emulator) Specialized Secure Platform
Monitor/Controller
Control Unit
Bio-Inspired Coordination Framework
Network-Level Defense (Dynamism)
Protocol-Level Defense (Dissemination)
Platform-Level Defense (Diversity)
> Topology Adaptation
> Node/Threat Isolation
> Dynamic Network Reconfiguration
> Intrusion and Anomaly Detection
> Peer-Feedback / Reputation
> Robust Protocols
> Randomized Instruction Sets
> Local Resource Monitoring
> Damage Detection

Figure 3. Proposed multi-layer infrastructure

As described in our quarterly reports, the coordination mechanism implemented over the three-layer architecture was based on the concept of Organic Resilience [13]. That approach was inspired by three core characteristics from biological systems, resilience and evolution:

- Multi-potentiation: Capacity of the biological systems to effectively change their functionality or physiological structure to assume different roles in order to address local or system needs.
- Redundancy: Using redundancy, complex biological systems are able to sustain some level of damage, while at the same time allowing them to perceive, identify and adapt to the problem.
- Feedback mechanisms: A key capability for biological adaptation is the multiple feedback mechanisms operating at different levels in the system.

Together, these principles are leveraged to enable the instantiation of the multi-layer defense. The feedback mechanisms borrowed from biological systems are represented by the information exchanged between the lower (detection) and middle layer (immunization). They provide signals related to the security event, and similarity information between services to enable the appropriate selection and re-constructions of affected services.

3.2.1 Online Learning, Detection and Classification

In the context of the work, traffic classification consists of the ability to identify a type, or a class of network traffic between applications, based only on its network properties, that is, without any pre-conceived knowledge about the source and destination applications, or their host operating systems. The underlying assumption is that network traffic can be observed at any point between source and destination and an assessment about its class can be made based on its properties

We included as an additional defense capability (as part of the immunization layer) the capacity to classify (and generalize) traffic patterns based on network protocols. As detailed in one of our paper publications [34], the goal is to allow agents to identify similar traffic patterns that could be correlated when enough evidence of an attack exists, or damage (i.e. the effects of an attack) is detected. Having pre-constructed clusters of similar flow patterns allows for the quick identification of important correlations.

For this purpose, we developed a Parallel Neural Network Classifier Architecture (PNNCA) for traffic classification. Considering standard web-traffic as an example, we sought to enable our algorithm to group similar request patterns, which in this example, would correlate to common websites.

Classification is done using the Parallel Neural Network Classifier Architecture (PNNCA). A PNNCA is made up of blocks of classifiers that work in parallel [14]. Figure 4 shows the architecture of a PNNCB. Each block is made up of a Radial Basis Function Neural Network (RBF NN). An RBF NN consists of three layers: input layer, hidden layer, and an output layer [15], and its output is calculated as:

$$y_i = \sum_{k=1}^{N} w_{ik}\phi_k(x, c_k)\ ,\ i = 1,2,\dots,m \qquad (1)$$

where, x is an input vector, N is the number of neurons in the hidden layer, w_{ik} are weights in the output layer and c_k are the RBF centers in the input vector. ϕ_k, is a Gaussian RBF function given as:

$$\phi = e^{\left(-\frac{(x-c)^2}{\sigma^2}\right)} \qquad (2)$$

The spread parameter σ, which controls the width of the RBF, is a very important parameter and directly influences the accuracy of the classifier. We have optimized the parameter to give a better classification and for improved feature selection.

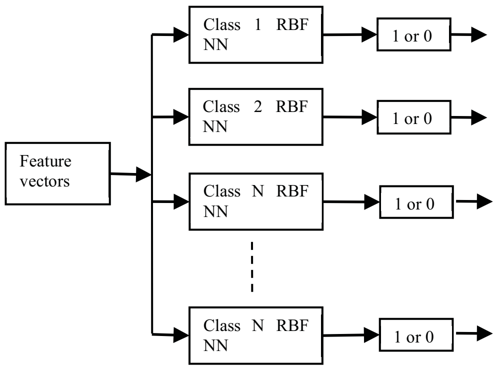


Figure 4. Architecture of a PNNCB

The number of blocks in the PNCCA is equal to the number of classes and the weights of each block are set during individual training. Each block of the parallel classifier is trained to tell if a class belongs to it or not.

The PNNCA uses the negative reinforcement learning algorithm (i.e., as it learns to identify packets belonging to that class, it also learns to reject packets not belonging to the class.). To measure the performance of the classifier, the Correct Classification Rate (CCR) is used along with a confusion matrix.

For web traffic classification, features are computed for each sequence of network packet [5]. The inter-arrival time information can be used to determine burst characteristics [5]. Packets are grouped together as bursts if their inter-arrival time is greater than a certain predefined threshold. The predefined threshold is set in such a way that it separates one set of flows or a session from the next. For example, if a user accesses a webpage three times, then the threshold should be able to separate the packets in three bursts.

Each session is described by a set of features. Previous studies have used a number of features to describe network traffic and classify them [7, 10]. The features are used to discriminate individual web traffic flows. A. W. Moore, et al. [3], described sets of discriminators for use in traffic classification. They use 248 features to define a traffic sample. Other researchers have used a smaller number of features because of the redundancy in the 248 available features. Runyuan Sun, et al., [7] used 22 statistical features to represent traffic; we have reduced that number to 17, shown in Table 1, because only forward traffic flows are considered in this study. Eight statistical features related to inter-arrival time and eight related to packet size information were computed as traffic features. The number of forward packets is the other feature. Inter-arrival time is the time difference between consecutive packet flows in a session. It is defined as:

$$time_{int\text{-}arr} = time_i - time_{i-1} \quad i = 1,2,3, \ldots n, \tag{3}$$

where n is the number of flows in a session.

Table 1. Features used

Flow metrics (duration, packet-count)

Packet size /payload information (mean, variance, median, minimum, maximum, 1st & 3rd quartile, inter-quartile range)
Total packets (forward flow)

With the features for packet sequences identified, we now collected the feature set from all the traffic observed in some point of the network. The features for sequences of packets are extracted into feature vectors that are fed to a neural network for classification. Figure 5 shows, for example, the set of normalized feature vectors for http sessions directed to a given website (www.google.com, in this example).

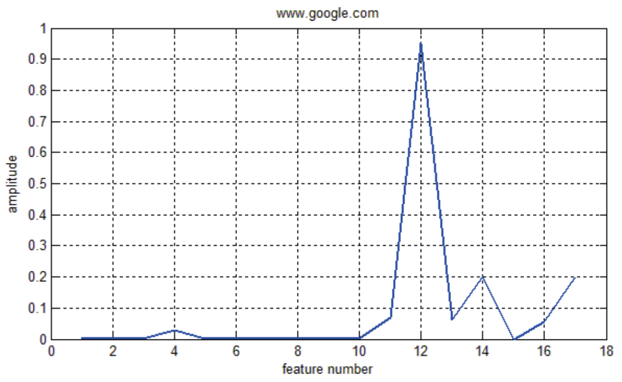


Figure 5. Example of feature vectors for a specific web-server

The traffic classification was demonstrated to classify HTTP sessions to specific websites (using simulated traffic), and results have been published in [34]. The approach was based on a supervised PNNCA to classify client sessions. In order to provide a preliminary evaluation of the proposed approach, we have used a set of well-known websites and have described the process for data acquisition. Used as part of an immunization defense layer, the approach could be used to identify SCADA-related traffic (to web-proxies and web-interfaces), even when hidden on alternative ports.

3.2.2 A Secure Execution Environment

Despite these excellent results, if the underlying platform analyzing the traffic is compromised, the system will not be able to operate without significant interference. As such, a key capability in distributed monitoring and control framework is the notion of attack (or damage) detection and robustness. Physical nodes or software components that are subject to failure or attacks will be quickly replaced by redundant elements; however, it is imperative, for the survivability of the system that such events (successful or not) can be prevented, and, if this is not possible, detected.

As illustrated in Figure 6, our approach for the development and demonstration of a robust platform was staged to create first a fully functional but limited prototype architecture. For proof-of-concept demonstration, some part of the infrastructure (for example the agent execution environment) may be external to the emulated secure architecture.

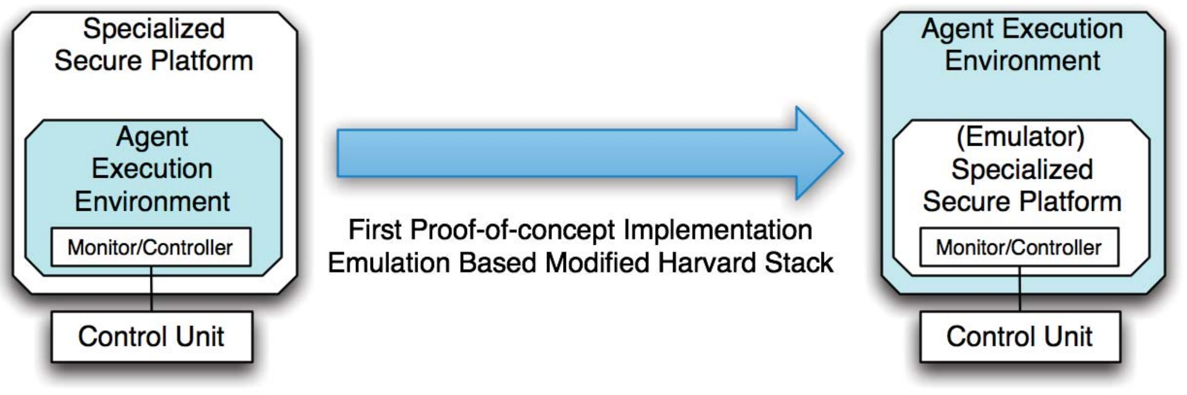

Specialized Secure Platform
Agent Execution Environment
Monitor/Controller
Control Unit
First Proof-of-concept Implementation
Emulation Based Modified Harvard Stack
Agent Execution Environment
(Emulator) Specialized Secure Platform
Monitor/Controller
Control Unit

Figure 6. Emulation-based staged design of the secure platform for the proposed architecture

3.3 Secure Host Architecture

A critical weakness in security systems is that the attacker has the ability to completely subvert a host. In such an instance, it may be impossible for code running on the host to detect that it has been compromised, yet the host (and its associated behavior) is wholly under the control of an attacker. Ways of accomplishing these attacks are numerous, but could even include running the entire host in a hypervisor, making detection deeply problematic [26]. Thus, for any control system, the security of the underlying platform is foundational to the security of the framework as a whole.

To date, there has been significant effort expended in the area of host security. Anti-malware software, host hardening, n-version programming, and host-based IDS have all helped buy down the risk of subversion, but ultimately, the underlying architecture of the host lends itself to compromise. Thus, in part of this project we have explored alternate host architectures that could be used to innately increase the security of the platform itself.

There has been a lot of discussion previously of the features of a secure host. In this section, we will provide an overview of the features that we have implemented and their benefit. Before looking at implementation, however, we will first examine the approach we have taken at the design level.

We have designed a hardware architecture that attempts to minimize vulnerability by design (that is, an architecture that has security properties innately). In an arbitrary order, the properties of our system are:

- Dual Stack Harvard Architecture
- Stack grows UP in memory
- Hard separation between processes
- Sparse population of opcode space
- Instruction Set Randomization
- Address Space Layout Randomization

- Carefully selected instruction set of fixed size
- Pointer Registration
- “Keyed” memory

The benefits of this approach are all aimed at making it more difficult for an attacker to inject code into the system or impact one process from another. In each case, the architectural choice provides “baked in” security, by blunting one or more attacker thrusts automatically.

While we would like to implement our device on silicon, such an undertaking is very expensive. Furthermore, hardware is not a good environment for design prototyping, especially at the early stages. Instead, we chose to create an emulator for the system. This actual implementation is outlined below, after a discussion of the benefit of our architectural choices.

3.3.1 Dual Stack Harvard Architecture

The idea of the Harvard Architecture is to directly isolate code from data. Thus, in a true Harvard architecture, it is very difficult (or impossible) to execute data. This effectively prevents simple code injection. However, it provides limited protection from a gadget-based execution approach, as described in [16]. The dual stacks do prevent the mixing of data and control flow on a single stack, but function pointers can still be used (in theory) to execute gadgets in code.

Our architecture is more accurately described as a *modified Harvard architecture*, as the kernel designates regions of RAM as code or data. This approach is similar to that taken by the *x86* family of processors, though the separation provided in our architecture is rather more rigid.

3.3.2 Stack Growth

Part of the problem with modern day computers is that the stack grows down in memory. This decision was not arbitrary, but was a deliberate architectural choice, made to conserve memory[1]. However, modern day machines are not (typically) highly memory-constrained, and the downward growth of the stack means that memory that is *higher* than that pointed to by the stack pointer is in use. Thus, if local variables are allocated on the stack, they exist below the return address that points to the calling routine (for an excellent – if simplified – overview of stack based buffer overruns, see [17]). Thus, if a buffer is overrun on the stack, it *always* overwrites memory that is in use. Conversely, if the stack grew upward in memory, a stack-based overrun would overwrite unused memory – and certainly not a control structure. Thus, an upward growing stack is inherently more robust than our current design, though somewhat wasteful of memory.

[1] The heap memory is allocated from the bottom up, and the stack from the top down. When the two meet, the machine is out of memory.

It would be trivial for us to modify the architecture of our emulator to provide additional protection against buffer overruns on the stack. For example, it would be easy to have the system issue a trap when new space above a "top of stack" pointer is accessed. However, we determined that the protection provided by the multiple layers of protection we have was sufficient.

3.3.3 "Hard" Process Separation

Modern operating systems typically are built upon rings of privilege; lower level rings have increasing privilege, until one reaches Ring 0, where the kernel resides. This ring has full access to all the memory of the machine. However, as pointed out by Bratus et al. [18], this approach may not be optimal. Once again, the design grew out of historical designs, and was intended to take full advantage of the limited processing power available at the time. Instead of thinking about a ring structure – that is, a *hierarchy* of privilege – our architecture focuses more on a matrix, where access is tightly controlled between all processes. Even the kernel does not have unrestricted access to a program's memory. This approach is reminiscent of the approach taken in Multics [19], where entering one privilege level means leaving another.

In practical terms, more rigid separation of processes is a double edged choice. On the one hand, it does mean that an error in one process is less likely to impact another. On the other hand, most utility computing devices use Inter-Process Communication (IPC) frequently. In our case, the environment makes this an easy choice. In a control system application, there is far less need for IPC than one might expect. As such, rigid process separation is a solid choice for our SCADA environment.

3.3.4 Instruction Set Randomization

The idea of randomizing the underlying instruction set is not new [20], and the technique is mature enough that the weaknesses of the approach have already been experimentally explored [16]. The underlying idea is that if the attacker does not know the instruction set, it is impossible to inject code. Our hardware architecture supports instruction set randomization natively, and is described more fully in the implementation section.

It should be noted that a Harvard architecture alone is not sufficient to defend from code injection attacks. For example, Francillon [20] demonstrated code injection on the Atmel AVR platform. Thus, this second line of defense to code injection is not in any way superfluous.

3.3.5 Sparse Instruction Set Space

The opcodes that represent valid instructions are sparse in our design – that is, if a particular opcode is chosen at random there is a minute chance that it maps to a valid instruction and will not generate an exception. In contrast, we have chosen a sparse opcode space, where most random choices will cause a segmentation fault in the machine, allowing a management process to decide how to handle the crash based upon context. Thus, the task of guessing opcodes is made more difficult.

3.3.6 Address Space Layout Randomization

Our system requires all programs to be stored on the system as pseudocode. This pseudocode is assembled at runtime to the actual machine architecture. As noted above, while this approach makes it difficult for an attacker to guess opcodes, it does not prevent the use of gadgets or return-to-libc attacks. As such, our system supports Address Space Layout Randomization (ASLR). While this is more of an operating system feature (rather than a native property) it is included here as it is a fundamental part of the architecture used to provide security.

3.3.7 Instruction Set Choice

Our instruction set (described below) was chosen to be simple but sufficient for general purpose computation. In addition, it was designed to make it difficult to carry out code introspection. For example, the only way to access the call stack is via special instructions; these instructions, and these instructions alone, can access stack memory. This type of protection again makes it more difficult for an attacker to take control of the underlying platform.

3.3.8 Registered Pointers

Our goal was to implement a C-like language for programming the platform. However, the concept of function pointers is implicit in C, yet this mixes code and data, forcing a compromise. To address this, the loader registers function addresses at load time. Only jumps and calls to these registered addresses are allowed by the system; any other call from a function pointer immediately causes a system trap.

Function pointers are used to access both internal and external functions by pushing parameters on the data stack and using a system call (`SYSENTER`) to jump to the function. Parameter passing conforms to the `cdecl` convention where the compiler generates code for the caller to clean up the stack, freeing the callee from that task. This allows us to limit the manipulation of process-owned code by the kernel.

For this Modified Harvard Architecture, calling functions from within program code is restricted to only those functions registered with the OS. A per-process table of function pointers is created by the OS at load-time from meta-data generated by the compiler and that table is used at run-time to validate and access functions.

A program file may contain functions that are only called from within that program or may contain functions that can be called from another program file. The compiler adds internal (non-exported) functions to the program file's symbol table and passes that table to the assembler/emulator along with other meta-data where it is available at the time of loading/linking. However, when a program module with exportable functions is created (such as a DLL), two sources of meta-data are needed.

The program calling the function lists all external functions it will call and each of the program modules that could potentially be called must provide a list of exported functions so that the linker can locate and load the DLL containing those functions and the OS can construct the function pointer table for the program file being loaded. A simplifying assumption is that all functions are exportable and that the compiler will generate a list of exportable functions (and their addresses) for all program modules.

The OS maintains an in-memory table of functions for the duration of the calling program's execution and the OS stores meta-data for shared libraries individually in the function pointer lists of each caller and does not maintain a global list of functions that are shared by all callers.

As such, function names need only be unique within the scope of the calling function (which is taken care of by the compiler) and not across all program modules. This also simplifies deleting function tables since it is not necessary to deal with a globally-registered function that has multiple users. However, if DLL sharing were implemented, unloading a DLL will require that the OS confirm that all users of that function have ended execution.

3.3.8.1 API for registering/accessing function pointers

The OS needs to have a simple interface for registering function pointers, calling functions and dealing with errors. As a start, the API for function pointers contains the methods described below.

- tablehandle CreateFunctionPointerTable() - creates an empty table of function pointers and returns a handle
- boolean RegisterFunction(String function_name, PTR function_pointer) - adds a function to the per-process function table, returns TRUE on success or FALSE if the function is already registered
- PTR GetFunctionPointer(String function_name) - gets the function pointer for the function with the specified name, returns that function pointer if successful or a NULL if attempting to access a function that is not registered
- boolean IsFunctionRegistered(String function_name) - returns TRUE if the specified function is registered and FALSE if not (implicitly called by GetFunctionPointer)
- boolean DeleteFunctionPointerTable(tablehandle the_table) - deletes the function pointer table

The following steps describe how the Emulator registers function pointers when the process is loaded and how it executes function calls.

- When the OS loads a new program module, it calls CreateFunctionPointerTable() to create an empty table to store the functions as they are registered.
- It then checks the program module's metadata for a list of the functions that the module could call. Using that list, it locates the libraries that contain each function and builds the function pointer table by calling RegisterFunction() for that function.

- A call to a function (by name) in the assembly language causes the OS / emulator to call the GetFunctionPointer() method and, if the result is a valid pointer, jump to that address.
- If a program module attempts to access an invalid (i.e., unregistered) function, the OS throws a TRAP and ends execution of the program module.
- When the program module ends execution, the OS deletes the table by calling the DeleteFunctionPointerTable() function.

With hindsight, it would have been valuable to implement an ARRIVE instruction, which marks a valid JMP point. This instruction would have allowed calls to proceed without involving the kernel. This change is something that we would like to implement in future designs of the system.

3.3.9 Keyed Memory

A common problem with existing computer software is the buffer overrun, where a buffer of allocated memory is filled with more data than it can hold. Even though we have made code injection difficult, a buffer overrun can still wreak havoc on data – for example, changing a critical value without authorization. Furthermore, an overrun on the data stack allows for the exploitation of function pointers; as such, their impact should be minimized by the hardware architecture itself.

Our system supports the idea of "keyed" memory. Thus, the processor contains a "key" register, akin to a segment register. All memory belonging to the system is given a key that can be rewritten (once) by the process which owns the memory. Thus, if a buffer is overrun, the wrong key will be in the key register. This causes a segmentation fault and passes control back to the kernel.

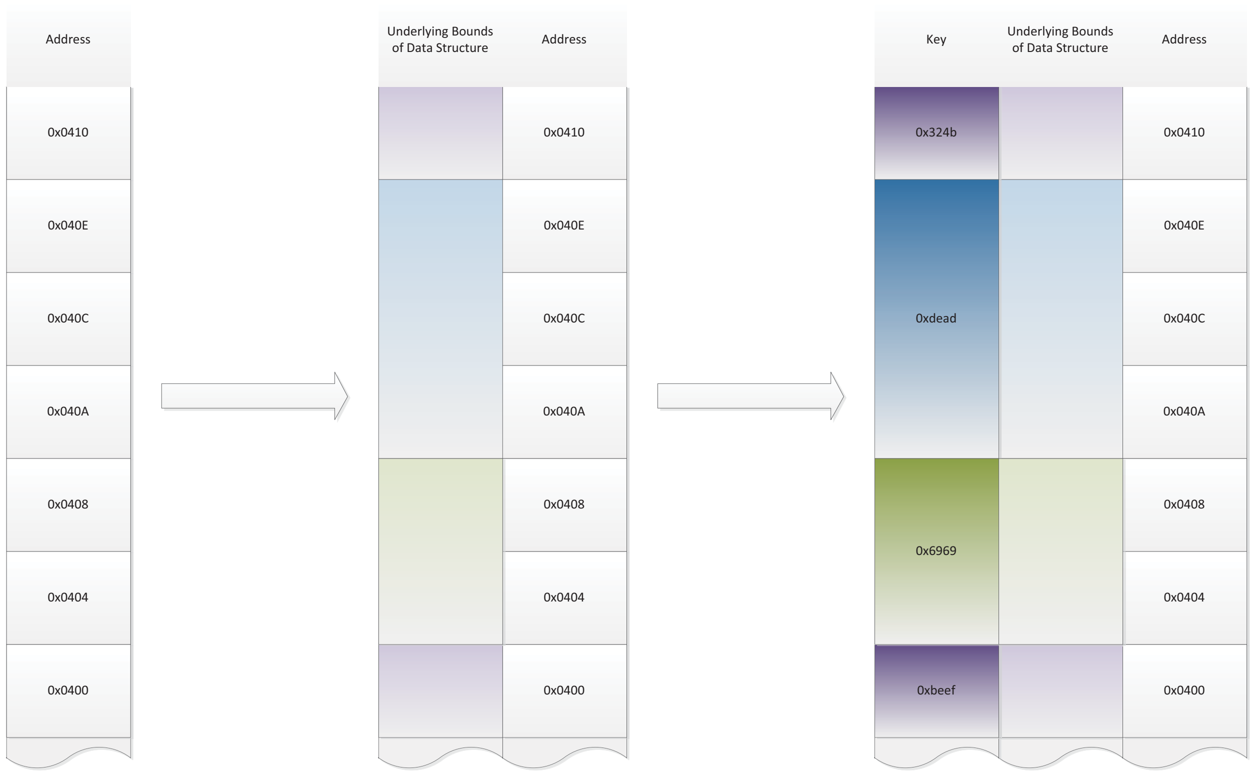


Figure 7: Memory Architecture that limits buffer overruns. The figure on the far left shows a traditional buffer in memory. The middle picture superimposes the actual bounds of structures. Finally, our implementation is shown on the right.

3.3.10 The System Boot Process

In order to understand our system, it is important to explore how the system is booted – after all, if the ISR occurs universally on the system, then the OS itself must be recompiled each time the system starts up.

The boot process is shown in Figure 8. Once the system boots it executes a post on self test. In turn, BIOS support is loaded, and a target architecture is loaded. This architecture is created leveraging a built in hardware random number generator (removing vulnerabilities related to inadequate random number generation). Such hardware-based systems are now readily available and affordable. The loader/compiler is then either recompiled or otherwise modified for this target environment, and the system restarts in the ISR environment. Once in this mode, the only way to return to normal operation is by restarting the system.

Within the new randomized environment, the microkernel of the system is compiled. This loads the supporting framework and user applications, before entering the master user runlevel.

Provided the original compiler and the box itself are secure from physical tampering, this boot process provides a self-contained way of entering the secure environment.

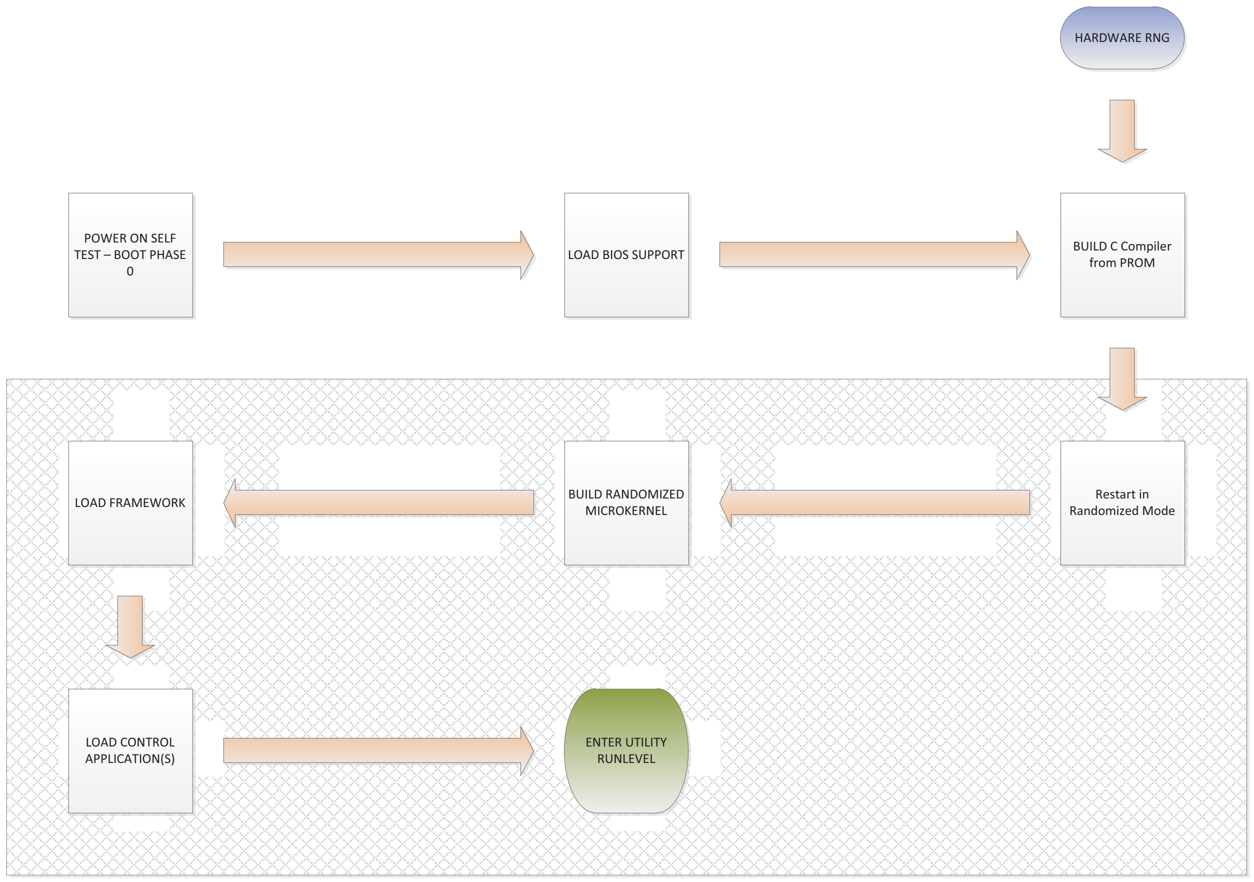

HARDWARE RNG
POWER ON SELF TEST – BOOT PHASE 0
LOAD BIOS SUPPORT
BUILD C Compiler from PROM
LOAD FRAMEWORK
BUILD RANDOMIZED MICROKERNEL
Restart in Randomized Mode
LOAD CONTROL APPLICATION(S)
ENTER UTILITY RUNLEVEL

Figure 8: The system boot process. Once the system is in the "hashed" area, the system is operating with a randomized instruction set.

3.3.11 The Emulator

As our architecture was experimental, it was prohibitively expensive to actually build the chip. Instead, we chose to build an emulator that could run the code produced by the cross compiler described below. Emulation is a reasonable approach, as the actual properties of the system could be investigated and the architecture changed quickly and easily. In the future, we would like to port our architecture to an FPGA platform, such as a Xilinx Spartan chipset. This would allow more study of the system, and also give others a better understanding of the practicality of the approach.

Emulating a processor is actually quite straightforward, and there exist several both open and closed source alternatives that were considered as a platform. However, we did not want to have to build and compile the entire operating system for the chip, and so chose also to emulate the features of the operating system in the interest of minimizing development time. However, it would not be difficult to port this functionality such that it runs natively on the emulated chipset.

The emulator is very simple in design. A chunk of memory is allocated to the emulator, and mapped to the addresses used by the emulated chip. The chip itself is emulated at the instruction level, and the emulator keeps track of register contents and flags. The emulator itself provides direct support for lower level OS functionality, such as process scheduling.

The emulated environment has the ability to load multiple processes at the same time. The scheduler uses a simple "round robin" approach to execution; OS features like this merit further examination, but suffice for a proof of concept. In addition to process support, the system also has a simple visual debugger. The syntax of the debugger is reminiscent of NTSD, and provides support for memory examination, break points and single-stepping of code.

The processor itself has the following registers:

Name	Function
EAX-AHX	General purpose 32-bit registers, which can be subaddressed (as AX, AH, AL etc.)
EIP	The instruction pointer. Cannot be manipulated except by a few instructions
ESP	Pointer to the control flow stack
EDP	Pointer to the data stack
PROC	Register which holds the current process ID
FLAGS	State from previous operations (for example, the Zero flag)
GS	The memory key which specifies blocks of memory

Figure 9: Short overview of the register structure of our processor.

As can be seen, the structure is very loosely based on the Intel architecture, but in addressing scheme only.

The underlying instruction set supported by the Emulator includes the following instructions:

- ADD
- AND
- CALL
- CMP
- DIV
- JA and JNA
- JB and JNB
- JE and JNE
- JG and JNG
- JL and JNL
- JMP
- MOD

- MOV
- MUL
- NOP
- NOT
- OR
- PUSH
- POP
- RET
- SUB
- SYSENTER
- XOR

Each instruction is fairly obvious in function based on the mnemonic, but the SYSENTER instruction needs further explanation. The SYSENTER instruction handles ring transitions for the system. The concept of SYSENTER is very simple, it provides functionality similar to the kernel calls found in common operating systems, but with greater isolation of resources and, therefore, increased security.

Based on the observations of Bratus et al. [18], the idea of an omnipotent kernel that works at the bidding of lower-privilege programs is not ideal for security. As such, calls to the kernel are carefully written to only allow for small areas of memory modification. This approach, which is based upon ideas found in Multics [19], requires that communication buffers are carefully controlled. As such, all calls to the SYSENTER command rely on the contents of the EAX register, which points to a buffer that is shared with this kernel. This buffer – and only this buffer – is writeable by the kernel function called. This approach, while not perfect, dramatically reduces the impact of a confused deputy attack, where a process with higher privilege is called to carry out an action that is forbidden for the calling process [22].

For example, the system allows calling functions by pointer using the SYSENTER command. The function mnemonic SYSENTER_WARP_THROUGH_FUNCTION_POINTER uses a string function name shared in the EAX-designated buffer to pass control to a function via a pointer. This function can be either a call or a jump, depending on the parameters provided. Thus, the SYSENTER command can provide powerful functionality while minimizing risk.

When a process is created, the system automatically creates a memory segment that is shared between the system and the running process. This segment is used to send data to and from the kernel and can also be used to pass data between processes, but only under the kernel's control. Enforcement is trap-based; that is, the SYSENTER handler does not have the rights to access other process' data, attempts to do so generate a trap. However, processes can modify memory permissions to allow other processes to access data in their shared memory segment. When processes request allocation of additional blocks of memory, they can specify whether those blocks should be unshared or a SYSENTER_SHARED_PAGE.

Each SYSENTER call requires that the EAX register contains the address of the shared segment. This segment is laid out as follows:

- DWORD containing a unique identifier for that type of SYSENTER
- The rest of the memory block contains call-specific information (parameters, constants, etc.) that will be used by the SYSENTER call

SYSENTER is used to perform typical process control functions, such as memory management and the execution of function calls. For example:

Process Exit - this call terminates the current process.

SYSENTER_PROCESS_EXIT

[OUT] DWORD - the process' return code. A return code of 0 indicates success, a non-zero return code indicates that an error occurred.

Malloc - allocates the amount of emulator memory requested, where the starting address is a random page.

SYSENTER_MALLOC

[IN] DWORD - the number of bytes to allocate

[IN] DWORD - a flag that indicates the type of memory desired, 0 requests normal, process-only data. Non-zero for a SYSENTER_SHARED_PAGE.

[OUT] QWORD - contains a 64 bit pointer to the starting address. The top 32 bits contain the memory key and the bottom 32 bits are the actual address.

Free - releases a section of memory that was allocated by the SYSENTER_MALLOC. An error code is returned for calls to free unallocated memory or memory that was not owned by the requesting process.

SYSENTER_FREE

[IN] DWORD - starting address for the section to be freed

[IN] DWORD - the key required to access that section of memory

[OUT] DWORD - an error code on failure or a zero on success

Grant Memory Permissions - modifies the permissions on memory a process owns, allowing it to be shared with other processes.

SYSENTER_GRANTMEMPERMS

[IN] DWORD - the address of the memory segment to be modified

[IN] STRUCT MEMPERM_STRUCTURE - the new memory permissions to be applied

Clear Memory Permissions - revoke all permissions on this segment, resetting them to the default of "process owner only"

SYSTENTER_CLEARMEMPERMS

[IN] DWORD - the address of the memory segment to be modified

Pointer-based function call - makes a function call by following a function pointer

SYSENTER_WARP_THROUGH_FUNCTION_POINTER

[IN] DWORD - a pointer to the name of the function to jump/call.

[IN] DWORD - a Boolean value -- if true, this function is a call and the execution stack will be adjusted to return to the call site. If false, this function pointer acts as a jump.

Sleep - puts the calling process to sleep for the specified time interval

SYSENTER_SLEEP

[IN] DWORD - the number of seconds to sleep

Another group of SYSENTER options handles I/O and communications, examples include:

Print String - prints the null-terminated string that is pointed to by the parameter.

SYSENTER_PRINT_STRING

[IN] DWORD - a pointer to the text to be printed, If the PTR is invalid, the calling process will generate a trap and terminate.

Create Socket - create a new, unbound datagram socket.

SYSENTER_CREATE_SOCKET

[OUT] DWORD - the socket handle

[OUT] DWORD – contains the error code, zero if successful, otherwise non-zero

Bind Socket - binds a socket handle to a specified interface and port.

SYSENTER_BIND_SOCKET

[IN] DWORD - the socket handle

[IN] DWORD - a pointer to a null-terminated string containing the host interface to be used

[IN] DWORD - the port number for the socket

[OUT] DWORD - the return code for this call, zero if successful, otherwise non-zero

Send To - creates a datagram socket and sends a datagram to the specified target.

SYSENTER_SEND_TO

[IN] DWORD - the socket handle

[IN] DWORD - a pointer to a C-string containing the address of the destination host

[IN] DWORD - the destination port

[IN] DWORD - the address of the data buffer to send

[IN] DWORD - the length of the buffered data

[OUT] DWORD - contains the error code, zero if successful, otherwise non-zero

Get Buffered UDP Command - retrieves a command string from a UDP port whose number is defined in the emulator configuration variables. (Defaults to 511). This instruction is primarily used for control messages between the emulator and external components, such as the Learning Component. The buffer must exist in a kernel-writable place -- i.e., a SYSENTER_SHARED_PAGE.

SYSENTER_GET_BUFFERED_UDP_COMMAND

[IN] DWORD - the address of the buffer where the command string will be stored

[IN] DWORD - the length of that buffer

Socket Close - closes the specified network socket

SYSENTER_CLOSE_SOCKET

[IN] DWORD - the socket handle to close

[OUT] DWORD - the error code, zero if successful

Finally, a test harness for the system was written in Perl. This test harness consisted of a number of test cases, where the output of the emulator can be compared to the output of known programs.

3.3.12 Effectiveness

Our approach is more than just Instruction Set Randomization (ISR), but at its heart, ISR is the foundation which underpins the security of the system. As such, understanding the effectiveness of our ISR implementation is critical to understanding the platform as a whole.

Perhaps the most comprehensive discussion of ISR and its vulnerabilities is provided by Sovarel et al., which introduces a set of attacks where the underlying instruction space is guessed by an attacking piece of malware [16]. Ultimately, their approach is based upon two properties of existing ISR implementations. First, the authors assume that the "encryption" technique used is susceptible to a fairly simple known plaintext attack. Based on previous implementations, this is reasonable, as XOR has typically been used to modify loaded instructions at runtime. Second, the authors assume that the system will simply remain static in the presence of many failed attempts.

While it can be argued that a system which rekeys itself each time an invalid instruction is detected is equally weak (for such a system, the attacker can remain in a fixed position, and simply wait for the system to "line up" with his attack), their tacit understanding is that the system allows itself to be attacked over and over again without taking any action.

Our system deals with the approaches outlined by Sovarel in several different ways. First, the dual stack architecture makes it hard for a buffer overrun to lead to a code injection attack at all. The function pointer registration also makes corruption of a pointer harder to forge into a workable attack. However, even assuming an attacker can mark some data as executable, the sparseness of the instruction space dramatically increases the number of attempts an attacker must make to determine the underlying instruction set. Furthermore, our system sends traps caused by "impossible" instructions to a higher level system (discussed above) that can reason about system behavior. Adding yet another layer of defense, our system deals with source code that is released in p-code (basically, assembly language) form, and so any unknown instruction is either a result of a bug in the loader or an attempt at code injection. Either situation is important, and should be trapped and reported upon.

Despite these good properties, our system is still potentially susceptible to return-to-libc type attacks. Fortunately, our pointer registration/jump registration design effectively prevents gadget based attacks (see [23]), but the legitimate libraries themselves can still cause a lot of problems for a defender. Once again, however, several features provide protection. First, our implementation of ASLR helps a great deal. Second, our definition of jump destination pointers helps dramatically. Third, our higher-level analysis of the system makes it difficult to attack the same system over and over again without adverse consequences. Finally, our compiler provides support for security, inherently.

This compiler-related observation provides the ideal segue to a new topic: the compiler itself. This is analogous to the way in which Sun thought about Java: security was provided both at runtime and at compilation time. As such, we now turn our attention to the main compiler used by the system.

3.3.13 The Compiler

Building the framework for our system is not a trivial undertaking. We must be able to compile standard C code into executable code that follows the specifications described above; further, the underlying binary implementation of this code should vary from machine to machine. In addition, data transferred between machines needs to be shared in a common format.

Rather than writing a new compiler, we chose to adapt an existing compiler framework to translate C code into the assembly language described above. We also had to create an I/O library to replace the standard C libraries that provide File IO, Keyboard Input, Video Output, and Network connectivity.

Our compiler employs the open source Clang front end [24] for parsing and syntax checking the source code and the open source LLVM library [25] for code generation. However, as described above, we modified LLVM so that the output from our compiler is assembly code that is executed in the emulator rather than executable binary code.

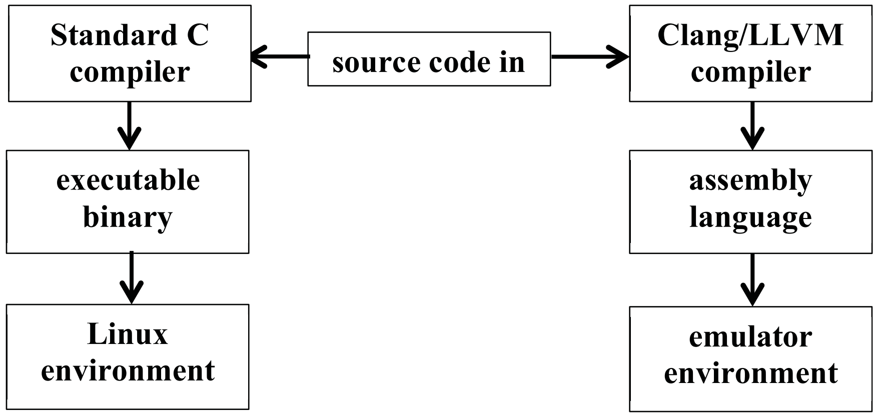


Figure 10. Compilation Steps

The compiler was developed in parallel with the emulator to ensure compatibility and to enable the compiler to inject assembly routines to support the I/O features mentioned above. Thus, the compiler replaces calls to C's standard I/O library (`stdio.h`) with calls to our own pre-assembled I/O routines, which the emulator then executes. Additionally, some of the security features require that the compiler inject extra instructions into the assembly code. For example, the Pointer Registration feature is based on the compiler inserting a SYSENTER instruction, with specific parameters, into the assembly to trigger the emulator's registration of function pointers. As a result, C programs could be written such that they could be compiled and executed natively in a Linux environment or compiled and executed in our emulator without any change to the source code, as illustrated in Figure 10.

4. System Characterization and Experimental Evaluation

Now that we have developed a system, our next task is to determine how effective the system is. Unfortunately, this leads to our next challenge: metrics for resilience are severely lacking. As such, we now turn our attention to the development of metrics for resilience systems in general.

4.1 Resilience Metrics Development

"Resilience" is challenging to define. The term refers to specific systems, tasks, outputs, and other conditions that vary between scenarios, which precludes the development of a universal metric that applies to all system in all situations. Just as different musicians cannot agree on the "best" rendition of a song, this existential definition of resilience has implications. In some disciplines such as ecology, the resilience of a system is defined as the time the system takes to recover to steady state conditions after a perturbation.

The considerations or resilience metrics that we explore in this section are particular to the context of cyber systems. Resilience of ecological systems, for example, rarely considers the magnitude of a response, focusing instead only on the time taken to return to pre-disturbance conditions.

Our notion of metrics is congruent with the extensive theory of measurement. As early as 1946, Stevens [27] proposed different levels of measurement, ranging from nominal, the labeling of objects, to ratio, the use of more sophisticated statistical techniques to determine equality, rank order, equality of intervals and equality of ratios. Typically, we consider measurements to range from weak to strong, with the weakest being nominal, and progressing though ordinal, interval, and ratio.

We would like our measurements to be as useful as possible. When we refer to resilience, we need to ask what a system being "twice as resilient" as another actually means. If this cannot be expressed in terms that are meaningful, the idea of a ratio-based measurement may be impractical or not applicable to the topic of resilience.

A *measurement* is a representation of a quantity. It is *not* the quantity being measured, and this is an important distinction. A measurement provides insight into the attribute under inspection.

This section proposes several guidelines for constructing metrics that are appropriate for a particular system, given our definitional ambiguity. Each consideration is described and then explored on an informal discussion.

4.1.1 Guideline A: All near-term metrics for resilience are likely to be ordinal

Engineers and scientists like to be able to assign numbers to things. "This GPU can carry out 1.1 teraflops—3 times as many as a CPU" is a meaningful statement that reveals something concrete about the systems under comparison. It is 1-dimensional, because it compares only computation speed. But resilience is not a 1-dimensional quantity.

Measuring the resilience of a system requires "rolling up" a time series $f(t)$ into a single number. Different system inputs produce different time series; loss of dimensionality creates a many-to-one mapping and, consequentially, a loss of information in the translation. Furthermore, the behavior (and recovery) of the system will vary depending on the failure or perturbation. For simple cases with a given set of possible outputs, it is typically possible to claim that one output is more desirable than another. This provides an ordinal metric.

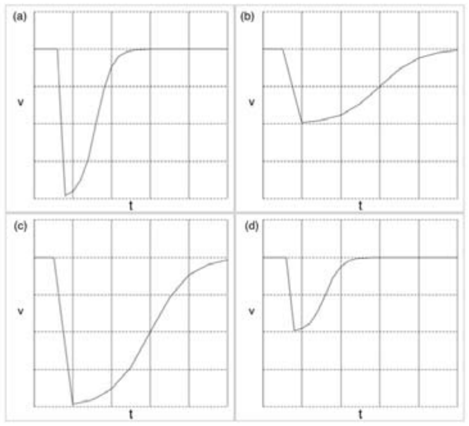


Figure 11. Examples or a system response to an external perturbation

Let us consider a very simple system as an example. We will use this example throughout to illustrate a system that is, at least at its core, very straightforward to analyze.

Consider a generator tasked to produce a certain voltage v. The system uses direct current (DC), so that we do not have to consider issues related to phase and timing; instead, we have a single scalar value that represents the system at any particular time. At time t_1, the system undergoes a perturbation of arbitrary origin. Figure 11 shows four possible graphs for the system response.

In panel (b) of the graph, the response curve is similar to that in (a) except that the magnitude of the response is greater for all values of t. We can thus argue that (a) represents a more resilient system than (b), but cannot really say how much more resilient until we consider the system in a given operational context. Also, panel (c) is similar to panel (a) except that the recovery happens more slowly. Panel (d) simply illustrates that curves may be unexpected and take arbitrary shapes—any measurement scheme must account for this possibility. Measuring the lowest point and time to recovery does not adequately characterize these curves. Similarly, the change in the area under the curve due to the perturbation, as proposed in [28], is only a partial measure of resilience.

Of course, we can measure quantities related to resilience. Time to recovery, for example, is a numerical measure of a single aspect of resilience [29]. Composing the measures of different aspects in order to reason about resilience itself is where our sense of ordinality originates.

Even with our simple example it is fairly easy to argue, by observation, that at least in vacuo, the system represented by Figure 1(a) is more resilient than those represented by Figures 1(b), 1(c) and 1(d). Recovery follows the same trend, it just happens more quickly. Even here, though, things are not quite that simple, and we will revisit these graphs in Guideline H.

4.1.2 Guideline B: Resilience measurements are particular to a particular perturbation

Different perturbations will cause different system responses. The failure and subsequent recovery of function of a particular part of a system is likely to lead to very different output patterns. This inherent notion of events in resilience has been noted not only in the security domain but also in the areas of organizational [31] and systems resilience [30]. In all cases, the concept relates to the challenge or disruption affecting the normal operation of the system.

Thus, when measuring resilience, we are actually measuring each individual perturbation and its different magnitudes, and providing an ordering that may be unique to a particular set of conditions. For example, one system might be resilient with respect to temperature increases of 5, 10, or 15 degrees Celsius, returning to steady state output after each temperature change. However, the same system may fail completely (that is, have no resilience whatsoever) in the event of a flood. For any system that we are likely to care about, there will be sufficient complexity that the resilience of the system will vary as a function of the type and magnitude of the perturbation. Determining the “best” system in this case will require an understanding of the disruptions that could occur in practice and of the users’ tolerance of them. Capturing this numerically will be difficult.

4.1.3 Guideline C: Resilience metrics are deeply dependent on the boundary drawn around the system

Rarely does considering the resilience of a single piece of a larger system in isolation make sense. Our example above (see Figure 11) considered a generator in isolation. If we increase the scope of that system to include what the generator powers, our determination of its resilience changes. Consider, for example, a generator that is powering a series of incandescent bulbs. Such a system is still usable when the voltage sags—the lights may dim, but still provide adequate lighting. In contrast, a generator powering a computing device will fail in its mission when the voltage sags below a critical value—the computer is either working or it is not.

Let us extend our example by providing support for a battery backup. When the generator output sags, the batteries can provide power for a certain number of kWh. For this system, the total power shortfall drives the failure—that is, it does not matter if the generator output drops to zero as long as it returns to functionality before the batteries are exhausted. Continuing to expand our view, suppose the batteries can run long enough for a human to intervene and install a new generator. The electronic system is not in itself resilient—it does not repair itself or recover—but the system *as a whole* is resilient (just not autonomically so).

The boundaries we draw around the "system" are critical in considering resilience. They must be carefully thought through and well defined. By changing the boundaries, a system that we considered not to be resilient may in fact be resilient, and *vice versa*. Any metrics we use to measure resilience are specific to a particular system boundary.

Perhaps the most fundamental distinction we can draw concerns human input discriminating between those systems with autonomic recovery, and those requiring some level of manual intervention. Determining which type of system we are exploring is critical to our choice of metric. In the case of an autonomic recovery, we expect the system to handle perturbations without human input. By considering humans as part of the system, *all* systems are in some sense resilient because the imagination and understanding of people provide an almost infinite pool of resources from which to rebuild the system. Conversely, for many real world systems, human intervention is a very real part of the larger system, and a resilience mechanism that provides adequate performance until humans can intervene is sufficient.

4.1.4 Guideline D: There is no universal way to combine multiple scenarios meaningfully to produce a "global" resilience ordering

As touched on above, any attempt to distill multiple measures of resilience into an "overall" measure of the system is fraught with problems. The different magnitudes of each class of disruption may have very different behaviors.

Attempting to unify the resulting curves into something that adequately represents the system is deeply contextual. Furthermore, when considering systems that need to be resilient to attack, any metric must take into consideration that a skilled attacker will attack the system at its weakest point. Attempting to reduce different aspects of resilience to a simple scalar loses so much information that we believe such a reduction to be ill-advised.

4.1.5 Guideline E: The ordinal ranking of a system could be different for each customer or application

The system requirements drive our ordinal ranking of resilience. Turning once again to our generator example, we can imagine two different sets of requirements. One customer may require the generator to maintain a certain minimum voltage at all times. Thus, any drop of voltage below this critical value makes the system as a whole not resilient even if the generator itself recovers. But another customer may care about the total time the voltage sags below its assigned value. If this sag lasts longer than a certain period of time, the system fails. So in this case, even if the generator capacity recovers, the system as a whole has failed.

This leads to two observations. First, the systems are no longer the same. External dependencies beyond the generator itself make the systems different, even though the generation component is the same. Second, the same behavior of the generator can be "good" for some customers and "bad" for others. Thus, we cannot treat the customer requirements as a black box. The resilience of the generation system itself matters less than the resilience of the system it supports. The customer requirements must drive our metrics for resilience; they are not tied to a single component.

4.1.6 Guideline F: Metrics for cyber systems are different than those of their physical counterparts

When we consider cyber operations—especially when we must account for the presence of a malicious adversary intent on damaging the system—we must think about systemic resilience differently than when thinking about random failure.

When dealing with random errors, it is possible to determine with some degree of surety the probability of the different failure modes of the system. As such, it is possible to consider the probability of different trajectories the system might take.

But, when we apply this reasoning to *non-random* failure such as an attacker might cause, a different picture emerges. A simple example is helpful. Assume we have 10 vulnerable web servers, and we patch 9 of them. We might conclude that, because we have repaired 90% of the machines, our risk has diminished dramatically. Alas, when facing an adversary, this is incorrect. Should the attacker identify the weak server, the site as a whole will be penetrated—so patching 9 web servers does not make the system as a whole 9 times more secure.

Thus, when dealing with an adversary, resilience needs to be viewed in terms of attacker capability and cost.

4.1.7 Guideline G: Considering just the system output is not a sufficient picture of resilience

When reasoning about resilience, it is important to measure the ability of the system to withstand further attacks [32]. For example, consider a system composed of n redundant generators. When a generator fails for any reason, it can be replaced by one of the other generators; conceivably, this could happen without any significant degradation of quality of service. However, after the failure, the system is not as resilient as it was previously. This “capacity” of the system to recover from subsequent failures is an important part of determining systemic resilience and is not necessarily captured by the output of the system until it actually fails. As such, an important part of measuring the system’s resilience is the cost in terms of its effect on the ability of the system to recover from subsequent failures or attacks.

4.1.8 Guideline H: Measuring resilience alone is usually not what we want

How we define “robustness” and “resilience” has strong implications when we try to compare the resilience of two systems. In particular, the sense that robustness is related to the system’s rigidity, and the sense that resilience is related to recovery, can lead to some counter-intuitive conclusions if we attempt to measure resilience alone.

Consider the system producing the graphs in Figure 11. Imagine a system that is not affected (in terms of output) by any event or disturbance—that is, the system essentially continues unperturbed by the attack or failure. Technically, this system displays robustness, but has not demonstrated resilience to a particular attack. Thus, one could argue that it could be less resilient than a system that is perturbed by, but recovers from, the same kind of event. In this scenario, measuring resilience may not make sense, at least from the perspective of recovery to an attack (as defined in [33]). An isolated measure of resilience may not be meaningful without a given context, and associated indicators of robustness of the system.

4.2 System Characterization and Application Scenarios

In order to test some of the concepts developed in this work we defined some application scenarios in simulation to illustrate the algorithms. Also as part of this work we have developed a sensing framework that was deployed at the IHMC building in Ocala, FL for environmental monitoring and control. The nodes were operational and implemented some of the defense mechanism outlined in our framework.

However, the small scale of the experimental network was very limiting for our tests and demonstrations. The proposed defense framework relies on biological analogs that depend on large scale, and reconfigurable systems. Because of that, we focused on simulated scenarios to illustrate our approach.

4.2.1 Application to Resilient Energy Management in Smart Grids Scenarios

Our initial developments and test implementation have been focused on a simple scenario of resource allocation in smart grids. In our view, this is a scenario that is general enough to support the development and test of distributed learning algorithms for resource allocation. At this point, the scenario is focused on resource allocation for cost reduction, but the mechanisms are just as applicable to our first defense layer – for dynamic network reconfiguration in response to attacks.

Our initial illustrative scenario is shown in Figure 12. A small feeder with renewable sources and standard loads (represented as houses) is connected to the grid. The renewable sources in this example are PV units.

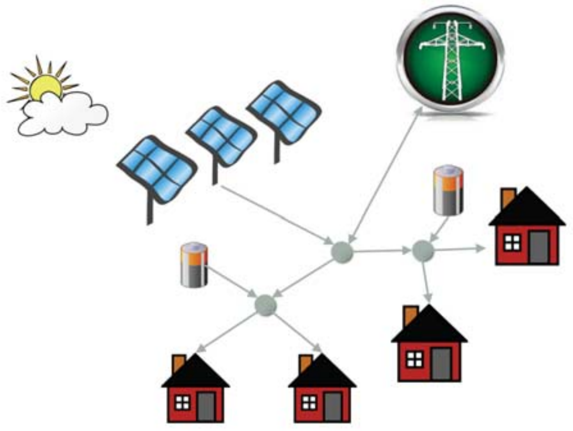


Figure 12. An illustrative scenario for test and evaluation of distributed learning components

Our scenario is designed so the renewable sources are capable of providing power necessary for the feeder. Energy is draw from the main grid only to compensate or fluctuations in the load and variations in power source. Variations in the renewable power source can be caused, for example by clouds, which leads to fluctuations in power and possibly energy draws from the main grid.

In this simplified scenario, our distributed agent system will monitor some of the meters in system and will try to compensate for power fluctuations by committing a few battery banks that are added to system for that purpose. Figure 13 shows a simplified diagram of the main components in our scenario, where houses represent loads.

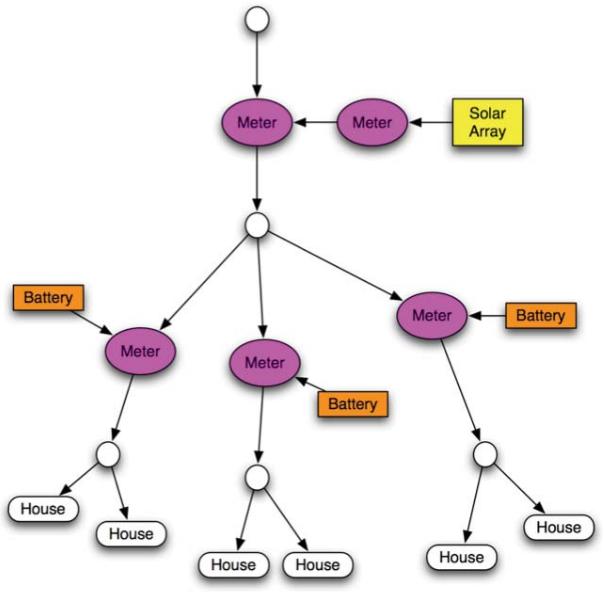


Figure 13. Simplified diagram of the illustrative scenario

Agents must be able to do that in coordination, and without a model of the feeder. They must be able to learn, from experience, how to optimally charge and discharge the batteries to minimize the use of external energy, and support the loads. Furthermore, for the defense of the infrastructure, agents must be able to learn and maintain a series of policies for different conditions that could be able to recover from an attack, as one of the three defense layers shown in Figure 13.

4.2.1.1 Distributed Learning for Automatic Unit Engagement

As illustrated in Figure 14, there are only five points in the system for agent monitoring and control, two of the main meters and three batteries. The coordination of battery sources is made by an agent located at the main meter, and the objective of that agent is to maintain the flow of energy through the main meter as close to zero as possible.

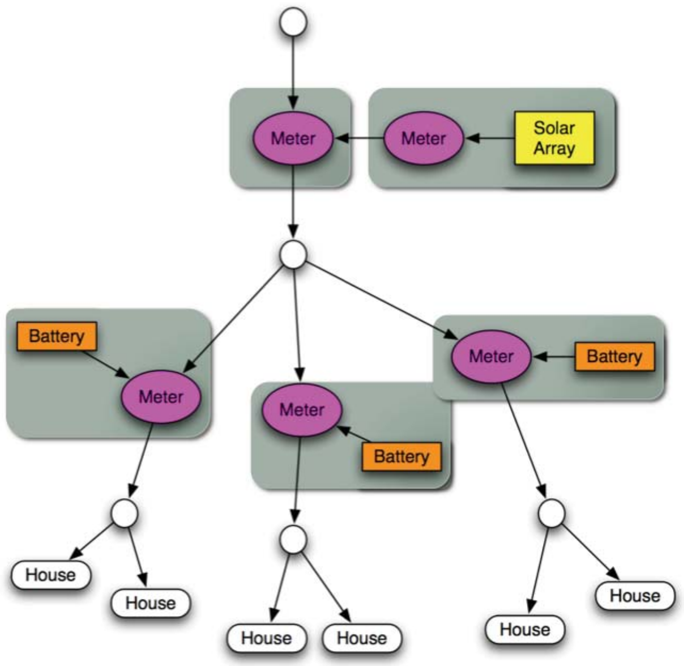


Figure 14 Distributed monitoring and control with five agents

Our scenario has been simulated using PNNL's GridLab-D environment, based on the Feeder R5-12.47.1, for a moderate urban area. The actual nature used in our simulations is show in Figure 15. In this example, agents use reinforcement learning strategy (Q-learning, to be more specific) to maintain the system. In this report, we will show only our preliminary results.

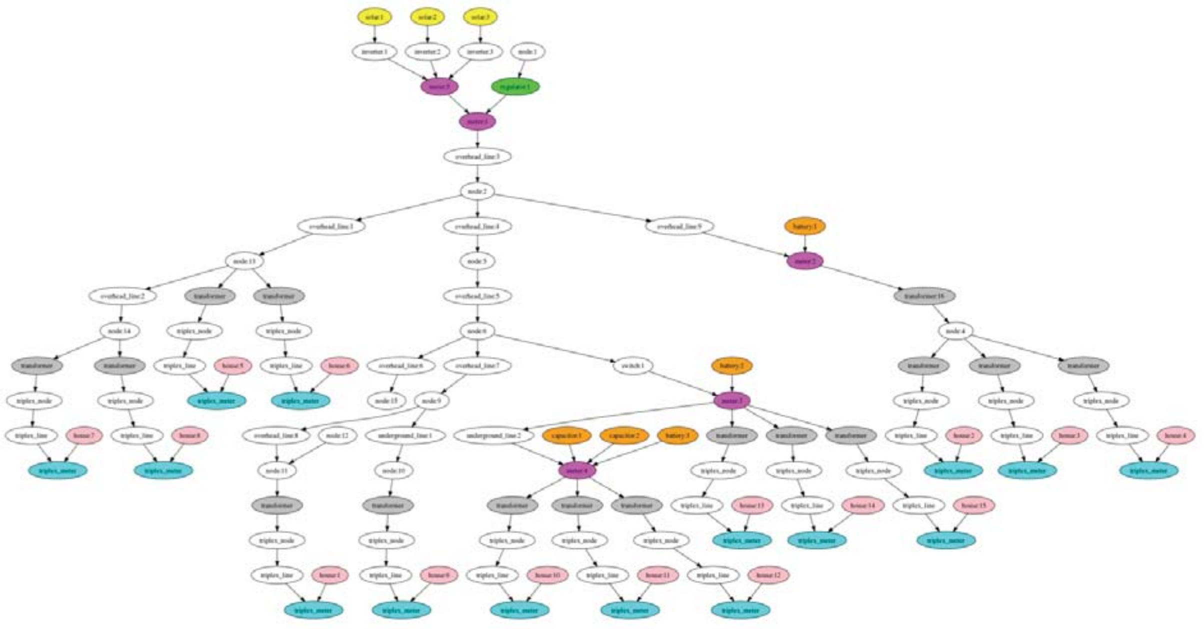


Figure 15. Actual simulation scenario in GridLAB-D

Our preliminary results for the multi-agent system coordination are shown in Figure 16. The graph shows the cost associated with the power flow through the main meter of the grid shown in Figure 15. Negative values power levels are indicated that spare energy from the PV source is being fed back to the grid, which is desirable. Positive values for the power indicate that energy is being drawn from the main grid to compensate for a temporary loss of power generation, which is not desirable.

The disruption shown in Figure 16 represents the cloud effects (that is, the drops in energy generated by PV sources due to clouds). Agents are responsible for detecting the effects and quickly identifying an optimal allocation of battery configurations (i.e. charge/idle/discharge states) that mitigates the problem. In this scenario, agents also take into account the location of the batteries in the system, choosing batteries that are close to the system loads at the time in order to minimize transmission power losses. Only information from the two main meters in the system (top two meters) and the state of the batteries is used for the control of the system.

In our simulation, agents interface with GridLAB-D in real-time, to gather the state of the meters and set the configuration of the batteries. The interface between software agents and the GridLAB-D simulation of the scenario show in Figure 15, is down through HTTP.

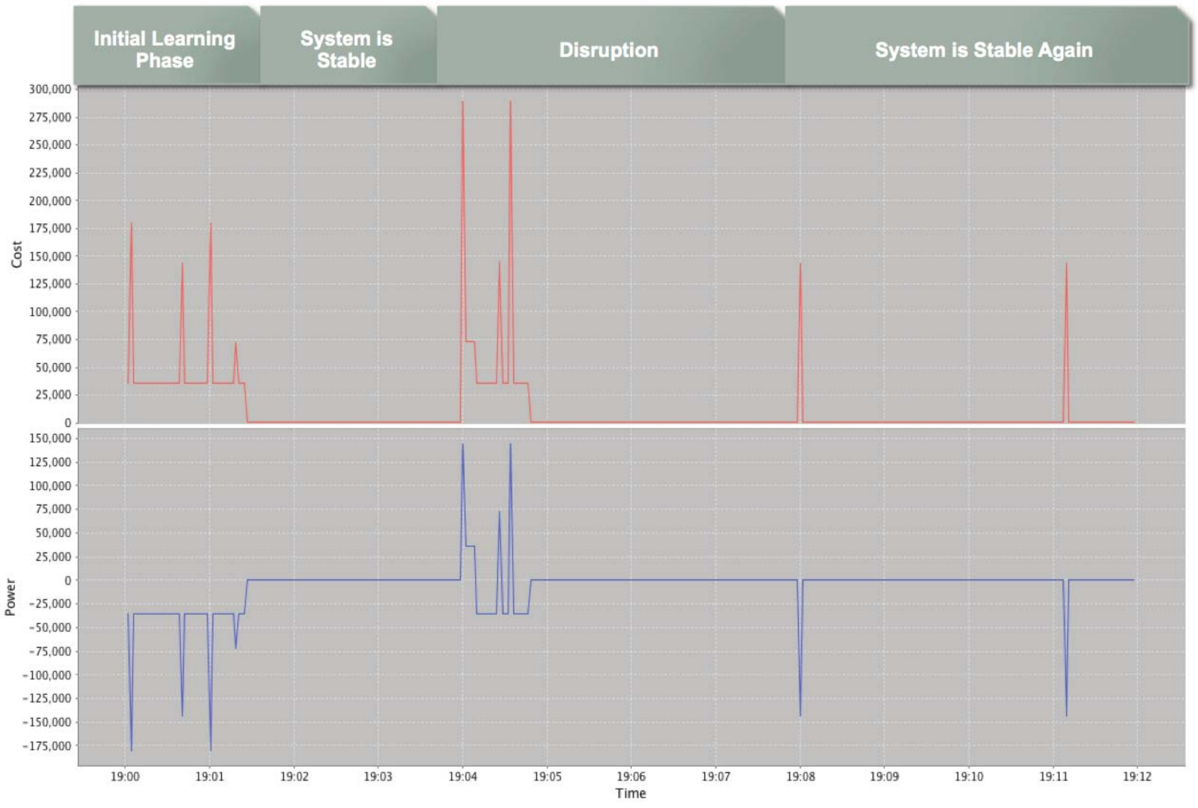


Figure 16. Initial simulation results, in GridLAB-D, for the unit commitment problem using multi-agent system

As part of the simulation agents must also choose the appropriate time to optimally recharge the batteries to maintain the resilience of the system. The strategy learned by the agents for each condition of the system is maintained as a set of optimal policies. In fact, agents maintain a policy probability distribution that can be used to response to each of the conditions experience by the system. This can be seen, in Figure 16, where at the beginning of the simulation, the agents take some time to identify the best (optimal) policy to charging/discharging the batteries based on the normal conditions of the system. Once policies are learned, the system operates at a stable condition (shown in the figure).

When a disruption occurs, the power measurement on the main switch becomes positive (i.e. drawing energy from the main grid), which changes the state and significantly increases the costs of the system. Agents again coordinate to find the new best set of battery configurations for the new condition. However, when the renewable source recovers, after the disruption, the agents no longer have to explore a new set of optimum configurations. They have learned from the system how to operate its unit sources under similar conditions are capable to move to stable condition much faster than at the beginning of the simulation. If a similar disruption (another cloud, for example), happens in the system, the agents would once again be able to react much quicker and respond to the problem, as policy sets would already exist for those conditions.

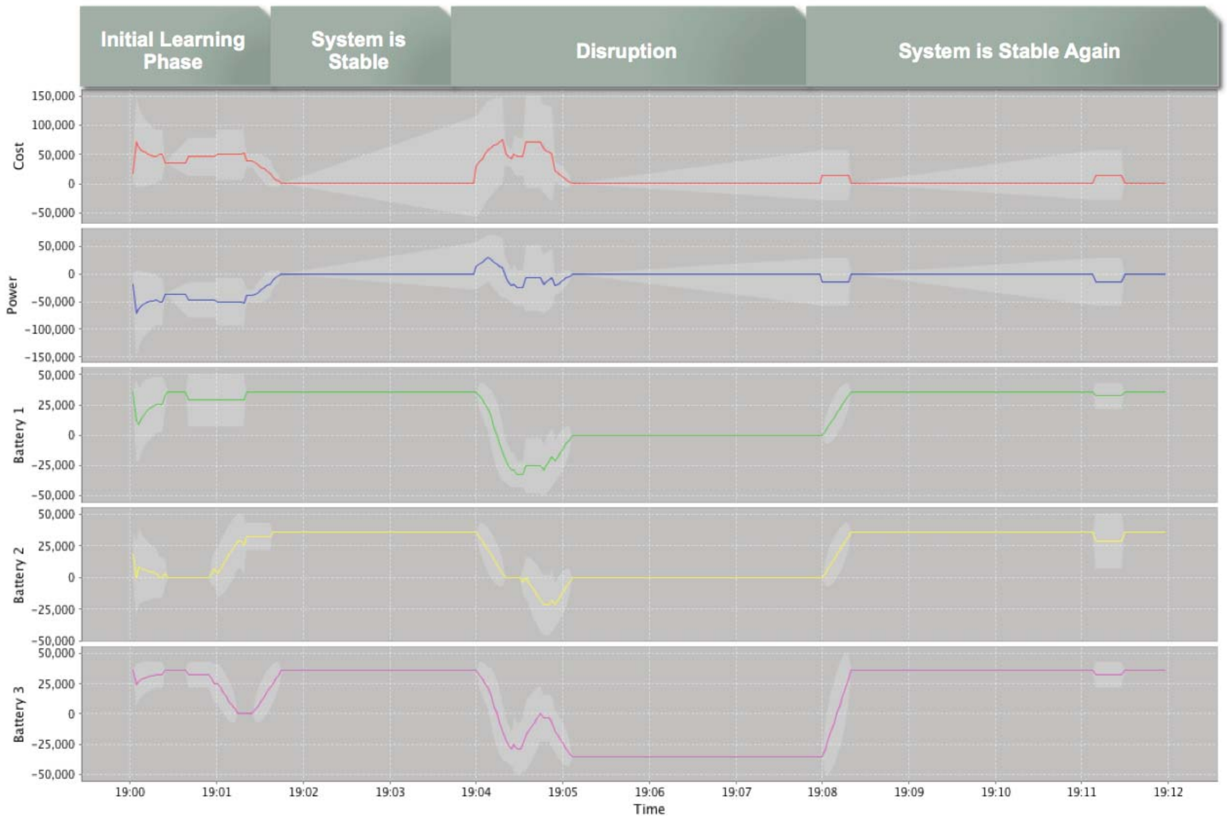


Figure 17. Smoothed version of our results, showing the variance on the agent-based solutions and also the state of the batteries.

Figure 17 shows a smoothed version of our results, where the variance of the solution is shown at each time. The figure also shows the state of each battery for the different conditions of the system. For the bottom three graphs, zero indicates that the battery is idle, negative values indicate that the battery is discharging (i.e. compensating for a drop in the PV source), and positive indicates that the batteries are charging. It is important to note that these initial results have been illustrated in terms of a unit commitment problem using battery sources, but the capability constitutes one of the layers of our proposed defense infrastructure.

4.2.2 Proof of Concept Application for Energy Management

For tactical operational settings, mission survivability is often defined as the ability to maintain the execution of a mission to its successful completion, even under unexpected adverse conditions, localized failures, or attacks.

In computational environments, a mission is often defined as an ordered set of tasks that must be performed in a timely fashion. We can create a parallel from this concept of mission in distributed computational environments to critical infrastructure protection.

A mission, in that context, would be defined as a composition of devices and services involved in the distributed instrumentation and control of large-scale infrastructures. Examples could include the distributed monitoring of an industrial plant, or a building energy management system.

To illustrate our approach, we have defined a simplified monitoring scenario, where the mission of the system is simply described as an ordered array of symbols. Each symbol represents a monitoring (or control) task carried by a device. The order of the symbols illustrates the interdependencies between tasks, for example, the parser function applied over a raw sensor response must be applied before the validation, storage and reporting.

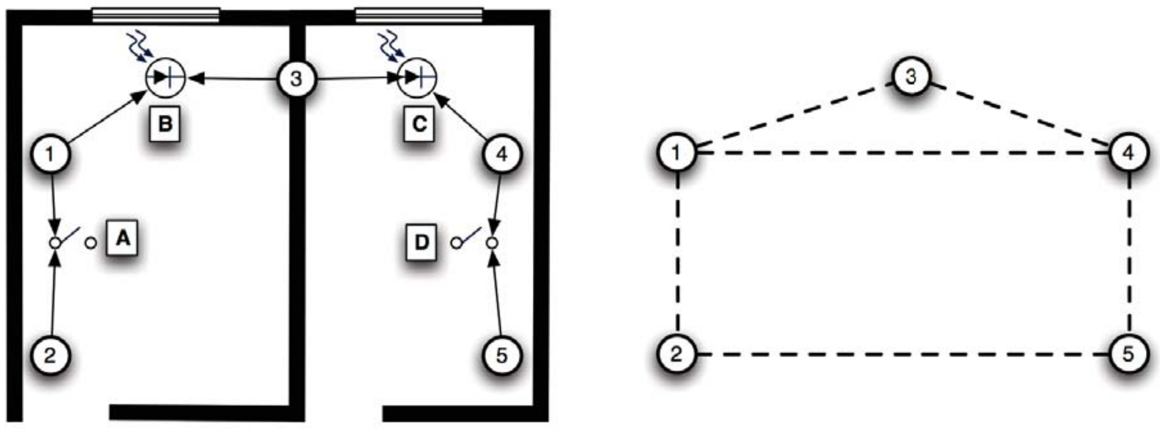


Figure 18. Simplified scenario illustrating the physical energy monitoring infrastructure.

Figure 18 illustrates a very simple version of the simulated scenario, with two rooms or offices, each of which has a light sensor and a light switch.

There are 5 nodes, and 4 possible actions: [A] open/close light switch of the first room, [B] measure the light intensity of the first room, [C] measure the light intensity of the second room, and [D] open/close the light switch of the second room. Node 1 is capable of measuring the light intensity and controlling the light switch of the first room. Node 2 is only capable of controlling the light switch of the first room.

Node 3 is capable of measuring the light intensity in both rooms. Node 4 is capable of measuring the light intensity and controlling the light switch of the second room. And finally, node 5 is only capable of controlling the light switch of the second room. This small network has enough redundancy only to handle single node failures.

Under the assumption that tasks can be spread across different nodes in the network (this will be further discuss later), Figure 19 shows an example of a mission in this scenario. The sequence of tasks is [B, A, C, D], which basically checks each light sensor and then opens or closes the corresponding switch. The nodes will automatically coordinate the execution of this mission between the nodes that can handle each task.

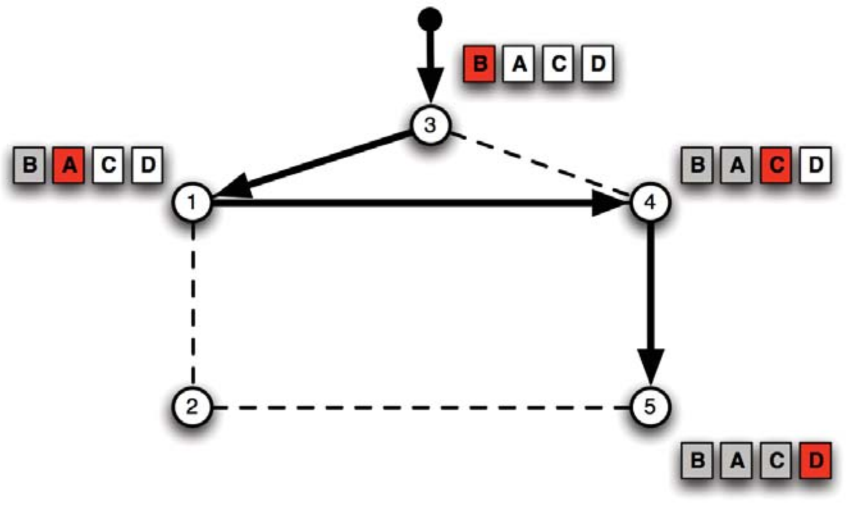


Figure 19. Example of a distributed mission execution in our illustrative scenario

The criterion for node selection is generally a cost function based on performance or resource utilization, constrained by policies, service availability, and other requirements. In this particular case, node 3 processes the first task (check the first light sensor), and then it forwards the mission to node 1, which then opens or closes the light switch and forwards again the mission to node 4. Once there, node 4 checks the second light sensor, and forwards the mission to node 5, which then executes the last task by opening or closing the light switch on the second room.

The node-level perception of service degradation and errors in the execution of a task is a critical capability for mission survivability. Nodes must be able to detect, isolate and identify a problem that may affect the performance of the mission. These tasks must be carried while maintaining mission execution with minimal disruption.

In order to accomplish that, a three component defense infrastructure will be used: (1) a component responsible for threat monitoring and detection, (2) a second component for temporary isolation by quickly re-allocating resources for mission maintenance, and a third component that bridges the identification of the threat with mission re-allocation, enabling a coordinated response to the attack and the dissemination of corrective measures to block future similar attacks.

Based on this environment, we will start the deployment of our proposed adaptation algorithm, for communications and service allocation. The algorithm is a variation of the approach proposed in [13], and will be further discussed in our next quarter. This scenario is preliminary, and represents a small scale of the actual simulation, which will build on a SCADA environment, developed from one of the test environments provided as part of the Gridlab-D simulation.

The point of creating this very simple environment with just a few nodes for energy management is that it can be mapped to our physical network deployment of sensors, and emulated secure host.

Combined, these two environments allow us to illustrate the proposed capabilities from the perspective of component interactions (i.e. SCADA devices, secure host environment, VIA infrastructure, and multi-agent system).

4.2.2.1 Experimentation Tested Environment for Resilient Energy Management

The unit connects to one of the USB ports in the router. Each router runs OpenWrt and has VIA installed. A module called SDCS in VIA automatically detects the unit (the LED in the box blinks 3 times when VIA connects to the Arduino board – see Figure 20) and starts receiving sensor (Temperature, Humidity, Light, Sound and Motion detection) data every 6 seconds.

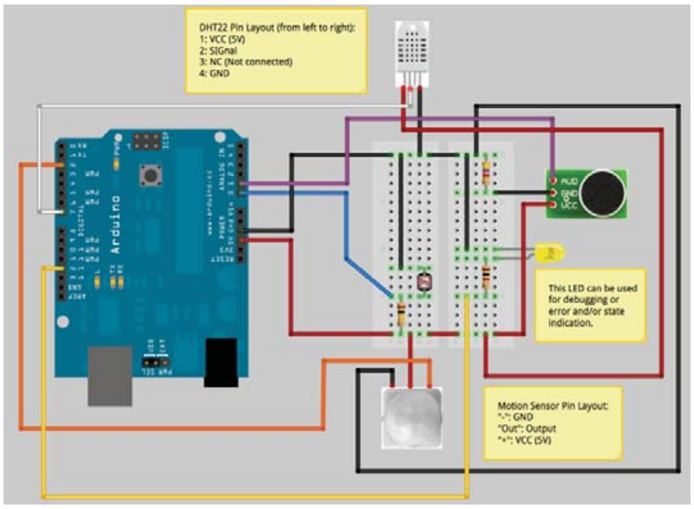

DHT22 Pin Layout (from left to right):
1: VCC (5V)
2: SIGnal
3: NC (Not connected)
4: GND
Arduino
AUD
GND
VCC
This LED can be used for debugging or error and/or state indication.
Motion Sensor Pin Layout:
"-": GND
"Out": Output
"+": VCC (5V)

Figure 20. Simplified Schematic of the board built for the experimentation nodes.

The sensor's data is disseminated using one of the "smart" flooding algorithms (Scalable Broadcast Algorithm, to be specific) through VIA, and one of the routers, which has been previously designated as the end-point, collects the data (including any data from a unit connected to it) and forwards it to a server using UDP. The server (written in Java) stores each sample with a timestamp into a database for further analysis.

Based on the designed schematics, we have built 8 nodes that can be easily integrated with the emulated secure host implementation for our test environment. The actual sensing element (shown in Figure 21) is connected to a small wireless router running VIA and the core agent system. Together, the system represents a single monitoring and control node in distributed control systems architectures.

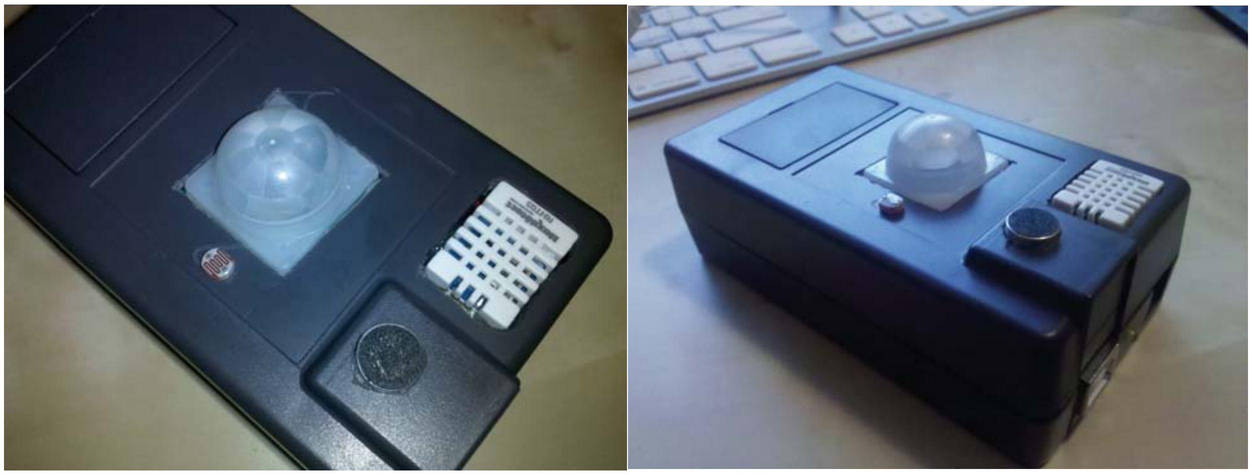


Figure 21. Surrogate Monitoring and Control Node

In practice the secure host platform would host all the applications running in the node (include the agent systems and the communications substrate) however, for test and demonstration purposes, our current implementation runs the monitoring and control software on a Linux-based ratio, which also hosts the VIA communications substrate and the agent system. The Emulated host resides between the communications/agent system and the monitoring and control application. This approach greatly simplifies the implementation and allows for the evaluation of the benefits, challenges and shortcomings of the proposed infrastructure.

While the emulation nodes have been implemented and tested for gathering and sharing information, they were not tested in connection with the secure host architecture. This task was not possible because the secure host architecture was emulated and not all the functionality to run the networking components of the sensor nodes were implemented as part of the emulation.

4.2.2.2 Large Emulated Environments

During the fourth quarter of the project a new scenario was developed with a larger number of nodes, and full emulation of the communications between nodes. The scenario is illustrated in Figure 22.

In the new scenario, links are simulated in NS-3. The simulations are executed over synthetic scenarios with grid topologies and with fully connected random topologies. The capabilities of each node will be distributed geographically in overlapping zones:

- nodes that are close together will have similar capabilities
- nodes that are far apart will most likely have different capabilities

Every node has a configuration vector (or a DNA, in our organic resilience formulation), represented as an *n-bit* long string. That string will define the combination of hardware, operating system, application and services running on the system.

In order to simulate the attack in our scenario, we define an attack as successful if a sequence $n - m$ of the bits of the victim match the signature of the attack launched by the attacker.

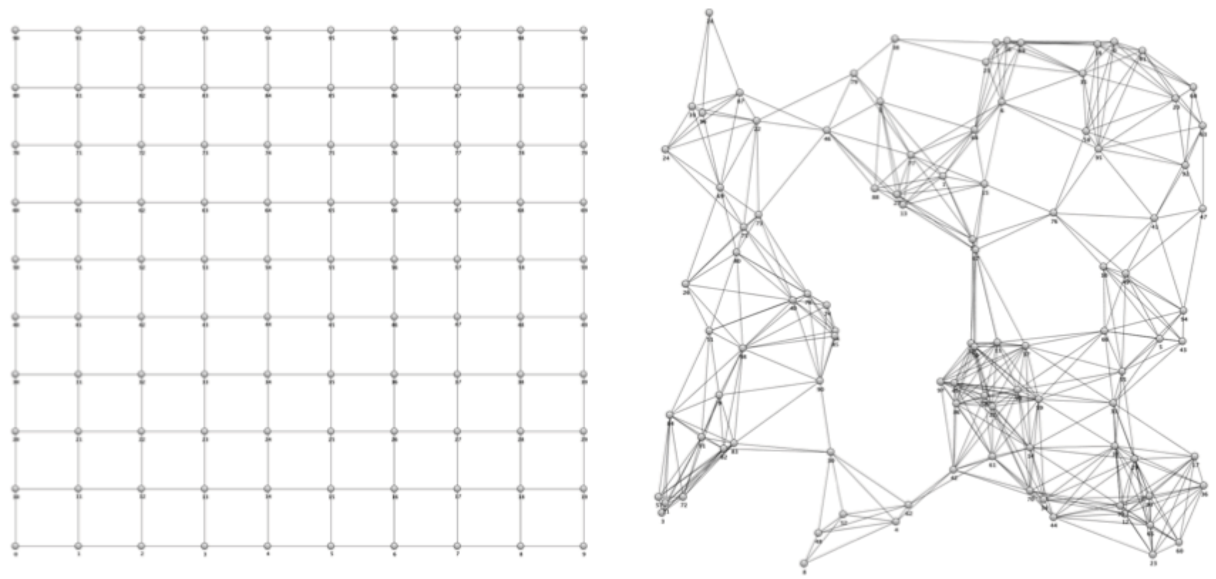


Figure 22. Survivability Simulation Scenario (multi-layer defense)

The objective of the defense infrastructure is to maximize the percentage of completed missions at any given moment. When responding to external attacks, the system is compared against two main baselines.

- a baseline with no attacks during the whole simulation.
- a second baseline with attacks but without any corrective measures.

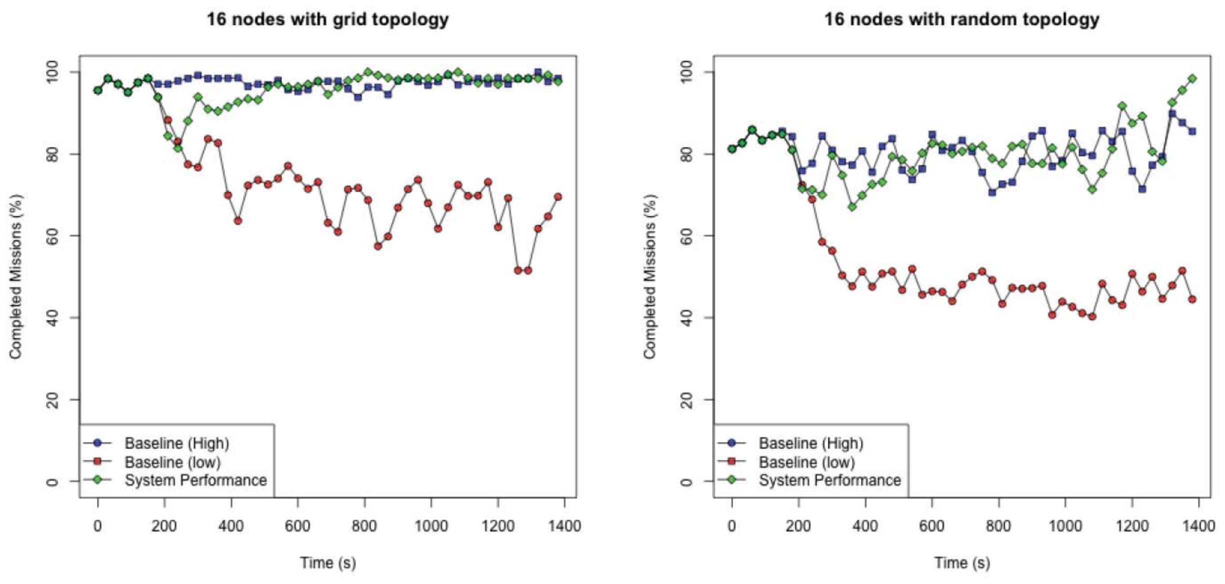
We have simulated both scenarios (using a random and a grid topology). A prototype implementation of the three layer defense framework was added to the simulation. The proposed approach was implemented and evaluated in NS-3 (allowing for larger scale studies). The simulations were executed over synthetic scenarios with grid topologies (Figure 22 - left) and with fully connected random topologies (Figure 22 - right), with 16, 25, 36, 49, and 100 nodes. The capabilities of each node were distributed geographically in overlapping zones, so nodes that are close together have similar capabilities and nodes that are far apart have different capabilities. The distribution of capabilities is important as it requires a re-allocation of services upon attack (one of the response mechanisms of the three-layer defense infrastructure).

In order to test the performance of the system we also defined missions (as individual tasks) that should be performed collectively by a set of nodes. In the context of the SCADA scenario, a Mission would consist, for example in a data capture from two sensors, the fusion (or aggregation) of the data, and the transmission of the result to a control node. The definition of mission is arbitrary, and is only used here to describe a set of functions to be completed by a group of nodes (or capabilities in the system).

Every mission-processing node has a configuration signature represented as a 4-bit long string. This "host signature" is just an analog of a biological "DNA" and describes the configuration of the node. For example, an IIS web server running on a specific version of windows would have a different signature then an apache server running on Linux, even though they were (in theory) functionally equivalent.

In our simulations, an attack is successful on a node if 3 of those bits describing its configuration signature, match the signature of the attacker, which is composed of 3 fixed bits and 1 wild card bit for matching. The analog in this case is the attack designed for a specific vulnerability of operating system and application.

The metric of interest for comparing results is the percentage of completed missions at any given moment of the simulation. The results were compared against a baseline with no attacks during the whole simulation. This baseline metric provides the upper bound for the operational limits of the infrastructure.



(a) 16-node scenario with grid topology (b) 16-node scenario with random topology

Figure 23. Simulation results of two topology configurations of a distributed system processing a fictional set of tasks

The results were also compared against a second baseline that illustrates the effects of the attacks without any corrective measures taken by the infrastructure. In this case, as more nodes become compromised, the overall performance of the system continually degrades. This second baseline metric constitutes the lower bound for the operational limits of the infrastructure. The attack detection happens indirectly (through the effects of the attack) and the identification happens by correlating the detection with the current state of the node (represented by the 4-bit configuration string).

The process takes some time, during which the services provided by the node are degraded. The response to the degrading attacks includes both a recovery and adaptation component. The recovery mechanism relies on restarting the compromised node to a previous safe image (read-only initial state). The adaptation component consists on an immunization capability that drives the mutations of re-instantiated servers to become resistant to previous attacks. Nodes identify a "mutation" strategy that is likely to make them less vulnerable to the same attack. For our simulations, the state of a node is represented by its 4-bit sequence and defines how vulnerable a node is to a given attack.

The immunization process involves having a node that has been attacked to announce its current bit-sequence to its peers, which drives "similar" nodes to mutate in order to become resistant to the attack. This is a simulation analog to a host that shared to its peers its configuration information upon detection of an attack, allowing peer nodes to estimate risk and, if possible, change its configuration and diversify.

Each scenario was executed 10 times with different seeds and the results were averaged across the simulation using a sliding window. The number of attackers and requesters in each scenario is equal to the square root of the total number of nodes, so simulations with 16 nodes had 4 attackers and 4 requesters, while simulations with 100 nodes had 10 attackers and 10 requesters. The requesters are randomly selected among nodes in the topology while the attackers are separate nodes that can attack any node in the network. The requesters issue a total of 400 missions each, each mission has 3 tasks and the tasks for each mission are randomly selected among the capabilities available in the network. Each task takes between 1 and 2 seconds to execute, but this execution time degrades when the mission is handled by an infected node. The missions have an expiration time proportional to the overall expected execution time for the whole mission, with some extra room for expected delays, like network related delays and scheduling delays. The attacks start after 200 seconds in the simulation and continue until the end. Figure 23 shows how the system performance evolves across the simulations.

Soon after the attacks start, the system performance is very close to the results of not having any corrective measures, labeled as *Baseline (low)* in the figures. But then it improves until getting close to the upper operational boundaries of the system, labeled as *Baseline (high)* in the figures. It is interesting to note also that for small topologies up to 36 nodes, the grid topology outperforms the random topologies when there are no attacks or when the protection system is active. But it is also noticeable that the random topologies show higher resilience to attacks when no corrective measures are taken for topologies of 25 nodes or more.

It would appear that the randomness of the topology provides some level of protection in itself, but it might be as well that the random topologies just have better connectivity allowing them to reduce network related delays due to multi-hopping. Another important point to notice is that the overall system performance degrades as the number of nodes increases. This happens because as the tasks for the missions are geographically distributed and as the tasks for the missions are randomly selected, with bigger topologies the chances of getting missions with tasks that are further separated is higher, causing higher network related delays on the execution of the missions, hence causing a higher number of missions to expire before completing execution.

5. Conclusions

We have published a number of papers describing our approach and some of the application scenarios. A list of the publications is listed as part of the references. Some adjustments of our activities were necessary through the course of the project. For example, the secure host architecture developed as part of this effort was done in emulation only; a natural extension of that effort (which was not in the current scope of the work) would be to implement that capability in FPGA to fully demonstrate the effectiveness of the design. We continue to pursue funding to leverage this work and continue hopefully to execute the port of the secure host architecture in FPGA.

Also regarding our application scenarios, due to the requirements of the proposed defense – we had to leverage simulations, rather than physical network devices emulating sensing and control nodes. However, we did create a small-scale physical version of the emulated network, to verify the functionality of our code in physical networks. A continuation of this effort would be to explore a large scale SCADA simulation (which was not available to us during the course of the work), and to create a hybrid simulation-physical environment that allows for the test of the proposed capabilities in real devices (possibly using a future FPGA version of the secure host architecture).

We have addressed the tasks in our statement of work, and hope to have contributed, through this research to advance the state of the art on the cyber defense of distributed systems and applications. Several of the lessons learned and technologies developed during the execution of this work will continue to be developed by our research group, as new funding – possibly from other sources, become available.

6. References

[1] Carvalho, M., Lamkin, T. and Perez, C., “Organic resilience for tactical environments,” in 5th International ICST Confernece on Bio-Inspired Models of Network, Information, and Computing Systems (Bionetics), (Boston, MA), December 2010.

[2] Yuan, S.; Chen, Q.; Li, P., (2009) Design of a four-layer IDS model based on immune danger theory, Proceedings of the 5th International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2009.

[3] Dasgupta, D. and Fabio Gonzalez, (2002), An immunity-based technique to characterize intrusions in computer networks, IEEE Trans. Evolutionary Comp. 6 (3), pp. 281–291, June 2002.

[4] Li, X., and Ye, N. (2001) “Decision tree classifiers for computer intrusion detection,” Journal of Parallel and Distributed Computing Practices, vol. 4, no. 2, pp. 179–190.

[5] Cho, S. and Park, H. (2003), “Efficient anomaly detection by modeling privilege flows using hidden Markov model,” Computers & Security, vol. 22, no. 1, pp. 45–55.

[6] Ourston, D., Matzner, S., Stump, W., and Hopkins, B. (2003) “Applications of hidden markov models to detecting multi-stage network attacks,” Proceedings of the 36th Annual Hawaii International Conference on System Sciences, p. 10.

[7] Abbes, T., Bouhoula, A., and Rusinowitch, M. (2004) “Protocol analysis in intrusion detection using decision tree,” in Information Technology: Coding and Computing, 2004. Proc. ITCC 2004. Intl. Conference on, vol. 1.

[8] Lardieri, P., Balasubramanian, J., Schmidt, D. C., Thaker, G., Gokhale, A., and Damiano, T. (2007) “A multi-layered resource management framework for dynamic resource management in enterprise dre systems,” J. Syst. Softw., vol. 80, no. 7, pp. 984–996.

[9] Carvalho, M. M., Pechoucek, M., and Suri, N. (2005) “A mobile agent-based middleware for opportunistic resource allocation and communications,” in DAMAS, pp. 121–134.

[10] Musman, S., Temin, A., Tanner, M., Fox, D. and Pridemore, B. (2010) “Evaluating the impact of cyber attacks on missions,” in 5th International Conference on Information Warfare and Security. Wright Patterson AFB, Ohio, USA: Air Force Institute of Technology, April 8-9, pp. 446–456.

[11] Sorrels, D., Grimaila, M.R., Fortson, L.W., and Mills, R.F., (2008) “An Architecture for Cyber Incident Mission Impact Assessment (CIMIA),” Proceedings of the 2008 International Conference on Information Warfare and Security (ICIW 2008), Peter Kiewit Institute, University of Nebraska Omaha, 24-25 April 2008.

[12] Grimaila, M.R., “Improving the Cyber Incident Mission Impact Assessment Process,” Cyber Security and Information Intelligence Research Workshop (CSIIRW 2008), Oak Ridge National Laboratory, Oak Ridge, TN, May 12-14, 2008.

[13] Carvalho, M., Lamkin, T. and Perez, C., “Organic resilience for tactical environments,” in 5th International ICST Confernece on Bio-Inspired Models of Network, Information, and Computing Systems (Bionetics), (Boston, MA), December 2010.

[14] F. M. Ham, K. Rekab, R. Acharyya, and Y.-C. Lee, "Infrasound signal classification using parallel RBF Neural Networks," Int. J. Signal and Imaging Systems Engineering, vol. 1, Nos. 3/4, pp. 155-167, 2008.

[15] F.M. Ham, and I. Kostanic, Principles of Neurocomputing for Science and Engineering. New York: McGraw-Hill, 2001

[16] Sovarel, A.N., Evans, D., and Paul, N., Where's the FEEB? the effectiveness of instruction set randomization, in: SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium, Baltimore, MD, pages 10--10, USENIX Association, 2005.

[17] Aleph One, Smashing The Stack For Fun And Profit (1996), in: Phrack, 7:49

[18] Bratus, S., Johnson, P.C., Ramaswamy, A., Smith, S.W. and Locasto, M.E., The cake is a lie: privilege rings as a policy resource, in: VMSec '09: Proceedings of the 1st ACM workshop on Virtual machine security, Chicago, Illinois, USA, pages 33--38, ACM, 2009

[19] Organick, E.I., The multics system: an examination of its structure, MIT Press, 1972

[20] Kc, Gaurav S., Keromytis, Angelos D. and Prevelakis, Vassilis, Countering code-injection attacks with instruction-set randomization, in: CCS '03: Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA, pages 272--280, ACM, 2003

[21] Francillon, A., and Castelluccia, C., Code injection attacks on harvard-architecture devices, in: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security, Alexandria, Virginia, USA, pages 15--26, ACM, 2008

[22] Hardy, N., The Confused Deputy: (or why capabilities might have been invented), in: SIGOPS Oper. Syst. Rev., 22:4(36--38), 1988

[23] Shacham, H., The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), in: Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA, pages 552--561, ACM, 2007

[24] clang: a C language family frontend for LLVM, http://clang.llvm.org/, visited May, 2012.

[25] Lattner, C., and Vikram, A., LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the IEEE Symposium on Code Generation and Optimization, pages 75--88, 2004

[26] King, S., Chen, P., Wang, Y., Verbowski, C., and Lorch, J., SubVirt: Implementing malware with virtual machines, in: Proceedings of the 2006 IEEE Symposium on Security and Privacy(314--327), 2006

[27] Stevens, S. S. On the theory of scales of measurement. Science 103, 2684 (1946), 677–680

[28] Wei, D., and Ji, K. Resilient Industrial Control System (RICS): Concepts, formulation, metrics, and insights. In Resilient Control Systems (ISRCS), 2010 3rd International Symposium on (Aug. 2010), pp. 15 –22

[29] Ives. A. R., Measuring resilience in stochastic systems. Ecological Monographs, 65(2):pp. 217-233, 1995

[30] Sugden, A. M. et al., 2001. "Resistance and resilience," Science, vol. 293, 7 Sept., 2001

[31] Westrum, R. (2006). A typology of resilience situations. In Hollnagel, E., Woods, D.Ashgate.

[32] Mendonca, D., Measures of resilient performance. In Resilience Engineering Perspectives: Remaining sensitive to the possibility of failure, volume 1 of Ashgate Studies in Resilience Engineering, pages 29-48. Ashgate, 2008.

[33] Bishop, M., Carvalho, M., Ford, R., and Mayron, L. Resilience is more than availability. In New Security Paradigms Workshop (NSPW) (September 2011)

[34] Mathewos, B., Carvalho, M., and Ham, F. M. Network traffic classification using a parallel neural network classifier architecture. In CSIIRW '11: Proceedings of the 7th Annual Workshop on Cyber Security and Information Intelligence Research (New York, NY, USA, September 2011), ACM.