A TRUSTED RECOVERY FRAMEWORK FOR SUPERVISORY CONTROL AND DATA ACQUISITION SYSTEMS

UN CADRE DE RESTORATION FIABLE POUR LES SYSTÈMES D'ACQUISITION ET DE CONTRÔLE DE DONNÉES

A Thesis Submitted to the Division of Graduate Studies of the Royal Military College Canada
by

Andrew Ivor MacKay
Lieutenant(Navy)

In Partial Fulfillment of the Requirements for the Degree of Master of Applied Science in Computer Engineering

September, 2018

Abstract

Supervisory Control and Data Acquisition (SCADA) networks provide control and monitoring of vital infrastructure. Increasingly, SCADA networks are composed of traditional IT network hardware and software. While this increases the risk of computer attack, it also means SCADA networks include embedded trusted computing technologies. This research leverages these embedded trusted computing technologies in order to design a Trusted Recovery Framework. This Framework is capable of restoring nodes in a SCADA network to a trusted state, that is, a state in which operators can rely upon them to behave as expected. The framework is resilient to both direct interference from an adversary on the same network and persistent malware on the node itself.

A proof-of-concept implementation of the Trusted Recovery Framework for the Integrated Platform Management System (IPMS), the SCADA network used by warships in the Royal Canadian Navy, was developed. This proof of concept implementation can be generalized to show that the Trusted Recovery Framework is a valid design for both for IPMS and SCADA networks in general. This research finds that the proposed design is capable of restoring SCADA network nodes to a trustworthy, operational state in the presence of malware.

Résumé

Les réseaux système de contrôle et d'acquisition de données (SCADA) assurent le contrôle et la surveillance des infrastructures vitales. De plus en plus, les réseaux SCADA sont composés de matériel informatique et de logiciels informatiques traditionnels. Bien que cela augmente le risque d'attaque informatique, cela signifie également que les réseaux SCADA incluent des technologies informatiques sécurisées intégrées. Cette recherche tire parti de ces technologies informatiques sécurisées intégrées pour concevoir un cadre de récupération fiable. Ce cadre est capable de restaurer des nœuds dans un réseau SCADA à un état de confiance, c'est-à-dire un état dans lequel les opérateurs peuvent compter sur eux pour se comporter comme prévu. Le cadre résiste à la fois aux interférences directes d'un adversaire sur le même réseau et aux logiciels malveillants persistants sur le nœud lui-même.

Une mise en œuvre de validation du concept du cadre de rétablissement fiable pour le système de contrôle intégré de plateforme (IPMS), le réseau SCADA utilisé par les navires de guerre de la Marine royale du Canada, a été mise au point. Cette implémentation de preuve de concept peut être généralisée pour montrer que le Cadre de Restoration Fiable est un concept valide pour les réseaux IPMS et SCADA en général. Cette recherche montre que la conception proposée est capable de restaurer les nœuds du réseau SCADA dans un état opérationnel et fiable en présence de logiciels malveillants.

For Vanessa and Tristan

Acknowledgements

I will begin by thanking members of the Royal Military College Computer Security Lab: Dr. Ron Smith, Dr. 'Sly' Leblanc, Dr. Vincent Roberge, Brian Lachine, Maj. Guillaume Vigeant, Capt. Jeremy Pacquet and Dr. Scott Knight. When I began this program I knew embarrassingly little (I still do) and I would not have been able to complete this project without the foundation you provided me.

The guidance and mentorship provided by Scott, my supervisor, was invaluable to both this project and so much more. Thank you for the latitude you allowed me to explore my own ideas and the trust you showed that I might end up anywhere fruitful.

I must express gratitude to my colleague LCdr. Robin Moll and my office mates Lorie Bell and Adm. Rubber Ducky. Sometimes all I required to solve an issue was a sounding board for my rants. Thank you for the patience and understanding you showed during the process.

And finally, thank you to my family and the innumerable ways you make anything possible.

Contents

Abstract **ii**

Résumé **iii**

Dedication **iv**

Acknowledgements **v**

List of Tables **ix**

List of Figures **x**

Abbreviations **xi**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Motivating Scenario 2

1.2 Statement of Deficiency 3

1.3 Statement of Aim 3

1.4 Research Activities 4

1.5 Thesis Structure 5

2 Background **6**

2.1 The SCADA Recovery Problem 6

2.1.1 SCADA Recovery Framework High-Level Requirements 8

2.1.2 Attacks on the Network Recovery Process 10

2.1.3 Methods of Malware Persistence 12

2.2 Threat Model 13

2.3 Bootstrapping Trust 14

2.3.1 Trusted Platform Module 15
Drawbacks of a Static Root of Trust for Measurement 17
2.3.2 Intel Trusted Execution Technology 18
2.4 Intel Management Engine 20
Intel Active Management Technology 21
2.5 Related Work 21

3 Design 25
3.1 Design Considerations 25
3.1.1 Trust Model 26
3.1.2 Assumptions 27
3.2 Design 27
3.2.1 High Level Architecture 28
3.2.2 Recovery Server 31
3.2.3 Node Software 33
3.3 System Operation 35
3.3.1 Initialize & Measure 35
3.3.2 Clone 37
3.3.3 Restore 37
Verification Phase 39
3.4 Design Summary 44

4 Implementation 46
4.1 The Integrated Platform Management System 46
4.2 Development & Validation Environment 47
4.3 Recovery Server 48
4.3.1 Network Boot Services 48
4.3.2 Data Layer 49
4.3.3 Application Layer 49
4.4 Node-based 50
4.4.1 Network Stage Bootloader 50
4.4.2 Trust Stage Bootloader 50
4.4.3 Remediation Environment 50

5 Validation 54
5.1 Analysis 54
5.1.1 Verification of the Proof-of-Concept Implementation 55
5.1.2 Validation of the Design 58
5.2 Performance Characteristics 58

6 Discussion & Conclusion **61**
6.1 Discussion . . . 61
6.1.1 Trusted Recovery Framework in the Context of a Wider Recovery Plan . . . 61
6.1.2 Limitations . . . 63
Use of `ntfsclone` vs. `dd` . . . 63
Trustworthiness of the Intel TXT DRTM . . . 63
Security Considerations of Enabled Intel AMT . . . 64
6.1.3 Future Work . . . 64
Extended Firmware Measurement . . . 65
Forensic Data Capture . . . 65
6.2 Conclusion . . . 65
6.3 Contributions . . . 66

References **67**

List of Tables

3.1 High level design requirements for a SCADA trusted recovery framework 28
3.2 The extent to which potential rootkit or bootkits are addressed 43
3.3 Attack scenarios and their implications on node availability and integrity 44
3.4 Fulfillment of the high level requirements by the proposed trusted recovery framework design 45

5.1 Trusted Recovery Framework Requirements (Repeated from 3.1) 55
5.2 Dell E6430 ATG Disk Utilization 59
5.3 Thinkpad T420 Disk Utilization 59
5.4 Time Required for the Restore Operation 60

List of Figures

2.1 Trusted boot sequence with UEFI firwmare [1] 17
2.2 Intel TXT late-launch sequence [2] 19
2.3 Network root of trust for tnstallation [3] 23

3.1 Trusted recovery framework architecture 29
3.2 Class diagram of the recovery server architecture 32
3.3 Design of Remediation Environment with supporting software . 34
3.4 Sequence diagram for the restore operation conducted by the recovery server 38
3.5 The verification phase control flow 42

4.1 Implementation of the remediation environment 52

Abbreviations

ACM	Authenticated Code Module
AMT	Active Management Technology
BIOS	Basic Input/Output System
CERT	Computer Emergency Readiness Team
CIA	Confidentiality-Integrity-Availability
CLI	Command Line Interface
CPF	Canadian Patrol Frigate
CRTM	Core Root of Trust for Measurement
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DRTM	Dynamic Root of Trust for Measurement
ENISA	European Union Agency for Network and Information Security
HTTP	Hypertext Transfer Protocol
ICS	Industrial Control System
IPMS	Integrated Platform Management System
LCP	Launch Control Policy
MBR	Master Boot Record

ME	Management Engine
MLE	Measured Launch Environment
NBP	Network Bootstrap Program
NFS	Network File System
NIC	Network Interface Card
NIST	National Institute of Standards and Technology
NTFS	New Technology File System
NVRAM	Non Volatile Random Access Memory
PCR	Platform Configuration Register
PXE	Preboot Execution Environment
RCN	Royal Canadian Navy
ROTI	Root of Trust for Installation
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
SCADA	Supervisory Control and Data Acquisition
SMM	System Management Mode
SRTM	Static Root of Trust for Measurement
TCG	Trusted Computing Group
TFTP	Trivial File Transfer Protocol
TPM	Trusted Platform Module
TXT	Trusted Execution Technology
UEFI	Unified Extensible Firmware Interface
UUID	Universally Unique Identifier
VBR	Volume Boot Record

1 Introduction

A Supervisory Control and Data Acquisition (SCADA) system is a class of Industrial Control System (ICS) which provides centralized data collection and management over dispersed nodes. Both the ICS Computer Emergency Readiness Team (CERT) and the European Union Agency for Network and Information Security (ENISA) have noted that SCADA networks are increasingly built using commercial-off-the-shelf computer hardware, IP protocols, and common operating systems and therefore now resemble IT networks at multiple layers [4, 5]. In addition, SCADA systems are increasingly internetworked with corporate IT networks which, in turn, are connected to the Internet.

These trends have increased the attack surface of SCADA systems and, consequently, the importance of robust incident response plans for SCADA systems. One key element of an incident response plan is the process through which a system is recovered. Despite its importance, the recovery process is given relatively little attention by SCADA cyber security standards and guidelines when compared to protection or detection measures [4, 6].

In the course of this research we examine advanced features in modern computer hardware which allow the integrity of a system to be verified remotely. We then present existing architectures which use these advanced features to generate trust in remote systems[3][7][8] and adapt their ideas to the design of a trusted recovery architecture for a SCADA system. The proposed trusted recovery architecture is then validated with a proof-of-concept implementation for a shipboard SCADA system used by the Royal Canadian Navy (RCN), the Integrated Platform Management System (IPMS).

1.1 Motivation

The National Institute of Standards and Technology (NIST) Cyber Security Framework provides guidance towards making computer network systems more secure. The NIST Cyber Security framework outlines five key activities inherent in managing computer system security [9]. These are: *Identify*, *Protect*, *Detect*, *Respond*, and *Recover*. Protection involves implementing measures to safeguard computer systems from identified computer security threats. These measures often compete with usability concerns. Consequently, for a network to remain usable, it will necessarily be exposed to malicious actors. Therefore, the NIST Cyber Security Framework also addresses the acts of detecting, responding to, and recovering from computer incidents. While detection methods are increasingly effective and the field of computer forensics is maturing, the discipline of recovery is still developing [10]. While traditional IT networks have the luxury of lower availability demands and are often supported by specialist IT staffs and therefore do not require as well defined recovery techniques, SCADA networks have high demands on availability and often don't benefit from on-site specially trained computer security personnel.

As a result of these factors, existing incident response approaches and system restoration software do not sufficiently address the unique operating environment and challenges of SCADA systems. For this reason, SCADA specific recovery approaches are required. While the recovery of SCADA systems is addressed by some SCADA cyber security standards and guidelines it is not treated at the same length, or to the same level of detail, as protection or detection measures [6],[4]. We argue that this indicates a lack of maturity in the field and the need for robust trusted recovery architectures for SCADA systems.

While the trend towards traditional IT technologies has increased the extent of vulnerability of SCADA networks, the modern commercial computer systems now in use also include hardware security features. These features provide immutable roots of trust in the computer hardware itself. These roots of trust can be leveraged in order to ensure the integrity of a computer node both locally to a user and remotely to a verifier over the network.

1.1.1 Motivating Scenario

The example of a SCADA network relied upon by a maritime vessel is illustrative. Modern warships rely on the complex interrelationships between propulsion, electrical, auxiliary, and damage control systems in order to float, move, and fight effectively. Control and monitoring of these systems is provided by SCADA networks. Like all SCADA networks, shipboard SCADA networks are designed to be highly reliable in the context of their environment but do not necessarily take asymmetric threats such as computer-attack into account. The Canadian Patrol Frigate (CPF), one of the warships operated by the RCN, is fitted with IPMS. While attempts to protect IPMS provide moderate assurance, the risk of computer attack can never fully be eliminated. The Stuxnet [11] worm demonstrated that isolation from the internet is not in itself a guarantee of security and ransomware such as WannaCry and Petya [12] have demonstrated how malware that successfully exploits a vulnerability in a legacy operating system can have a devastating effect across the numerous networks. These represent real world examples of malware which pose a direct threat to systems such as IPMS.

The personnel responsible for operating and maintaining IPMS are experts in the field of maritime engineering. However, a CPF does not deploy with personnel trained in computer forensics or computer incident response, the two disciplines required to reliably recover a system following a compromise. The situation is further complicated by the fact that IPMS is constantly relied upon to control vital ship systems and conduct damage control operations. This means that a prolonged loss of availability could potentially be catastrophic to the ship and its mission. While there are commercial products which support the timely recovery of remote systems in a network, they do not address an adversary who may still have presence on the network, or low level persistence on a node. As a result, in the event IPMS is compromised the RCN is incapable of restoring the system to a trustworthy state. This means that while there may be methods to restore functionality in IPMS, there is currently no way for the crew to resume confidence in IPMS following the detection of a network compromise.

1.2 Statement of Deficiency

The motiviating scenario above is a particular example of the general case, that the recovery of SCADA systems is often conducted on an ad-hoc basis. Many operators have little understanding of how to recover systems within SCADA networks effectively and the extent to which potential methods will address persistent malware. There is currently no solution which can conduct a fully automated trusted recovery of a SCADA network in the presence of malware.

1.3 Statement of Aim

The aim of this research is to investigate the design of a framework which leverages embedded security features in modern computer systems in order to conduct an automated and trusted recovery of a SCADA system following a computer security compromise. Trusted, in this sense, means that following recovery, the SCADA system can be relied upon to behave as expected without the influence of malware. This framework is constrained by fact that SCADA network nodes are typically dissimilar and therefore a recovery procedure must preserve the particular software and configuration of each. Potential frameworks are also constrained by the fact that SCADA network operators are generally not security experts and are unable to conduct comprehensive forensics investigations or tailor made recovery operations. As a result of this lack of expertise, any framework that supports recovery of SCADA networks must automate the process. The IPMS network is used as a representative SCADA system for validation of the approach. Like most modern SCADA networks, IPMS is built on modern computer hardware which provides a number of embedded security capabilities which could be leveraged to create a chain of trust from a trusted root in hardware to an operating system's run-time environment. They also include advanced remote management capabilities which allow an administrator to interact with a system remotely without relying on software tools or the network stack on the potentially infected operating system.

1.4 Research Activities

In order to meet the aim of this thesis various embedded security technologies were studied. The research involved an investigation into how these technologies could be incorporated into the design of a Trusted Recovery Framework. The two principal thrusts of this research were the design of the Framework and the implementation and validation of the framework for the target network.

The design of the framework began with an inquiry into the characteristics of SCADA networks which influence potential recovery frameworks. Following this a threat model was developed based on a credible threat faced by many SCADA networks as well as common threat models used in the literature when evaluating trusted computing technologies. From both the SCADA networks characteristics and the adversary model, a list of high-level requirements and design constraints was developed. With this list in mind platform embedded security capabilities were incorporated into the design for a Trusted Recovery Framework.

The Trusted Recovery Framework design was implemented for the target network, IPMS. Both the implementation and validation were conducted using a test-bed network which included actual IPMS hardware and software.

The research investigation began with experimentation with trusted computing technologies as well as associated technologies required to conduct the recovery of a node. The results of this experimentation were used to inform a novel design and implementation of a process by which a single node could be recovered in a trustworthy manner. The second phase of the research investigation involved the design and implementation of a Recovery Server which supports the restoration process on multiple nodes as well as acts as an external verifier of the integrity of the recovery process. Once the implementation was complete, it was validated against the list of high-level requirements for the Trusted Recovery Framework. Through this testing, the proposed Trusted Recovery Framework was validated, both for IPMS and for any SCADA system with the same generic characteristics.

1.5 Thesis Structure

The remaining chapters will outline this research in greater detail. Chapter 2 provides a thorough discussion of the recovery problem as it pertains to SCADA networks. It also contains an enumeration of the threats posed to the recovery process by an adversary and presents the threat model used in this research. Chapter 2 also includes the background necessary to understand the embedded security capabilities used in the proposed design. Chapter 3 presents the high-level architecture of the Trusted Recovery Framework and explains the details of its operation. Chapter 4 details the proof-of-concept implementation for IPMS, the target network. Chapter 5 provides an analysis of the proof-of-concept implementation in order to validate the proposed design. Chapter 6 discusses the limitations of the design as well as potential follow on work to this research before concluding the paper.

2 Background

In this chapter research and technologies applicable to the problem of recovering SCADA networks following a network intrusion will be discussed. As a result of the unique composition and operating conditions of SCADA systems the process by which they are recovered from a compromise differ from that employed by most IT networks. These differences will be expanded upon in the first section of this chapter where we describe SCADA systems in more detail as well as expand upon the problem of recovery. The following two sections discuss the threat posed to the recovery process by malware. This includes a discussion of how malware can subvert the recovery process or avoid it completely by persisting at a low level within the node. This will be followed up by a section which describes trust in computing systems and how modern hardware security features can be leveraged to achieve trust. Once this technical understanding has been achieved relevant work in the field will be reviewed, including academic research which could be adapted to the problem of recovering SCADA systems.

2.1 The SCADA Recovery Problem

ICS-CERT and ENISA have both noted that SCADA systems are increasingly similar to IT networks[5, 13]. This similarity includes the use of industry standard computers, operating systems, and network protocols. While the adoption of IT practices can decrease costs and increase capability, it also increases the attack surface. The threat is increased further as SCADA networks become more and more interconnected with corporate IT networks. This factor, along with the fact that SCADA systems are often responsible for the safe operation of critical infrastructure, creates an acute need for SCADA system security.

While SCADA networks can resemble traditional IT networks on some levels, their unique operating conditions make the task of administrating them drastically different. Some of the key differences between SCADA networks and IT networks include geographic distribution of nodes, operational context, and the importance of availability [4]. These differences are important in the development of SCADA security tools as they give rise to special considerations.

The problem of recovery following of computer intrusion is one example of this. Many IT networks can tolerate a period of downtime to allow for the network to be reconstituted following an intrusion. In fact, as advised by NIST [8], organizations should conduct business continuity planning to ensure operations can still be conducted while sections of their computer network are quarantined and a CERT conducts response actions. In some cases, the victim systems may be kept operational to allow counter intelligence surveillance activities to be conducted against the attacker [14]. These courses of action are desirable because in the traditional Confidentiality-Integrity-Availability (CIA) triad confidentiality and integrity concerns often outweigh availability. This property does not hold in SCADA networks. As SCADA networks are responsible for the safe and efficient operation of physical machinery their availability is of the highest concern. Following this, integrity is a secondary concern and confidentiality is the least pressing requirement [6].

The ICS-CERT publication, *Developing an ICS Cybersecurity Incident Response Capability*, addresses the issue of SCADA system recovery [13]. Incident Response encompasses all the activities an organization will undertake in response to a compromise. These include containment, remediation, and system recovery. Containment involves stopping both the spread of malware and any damage caused as a result of the compromise. Remediation is the activity of removing the malware and returning the node to a trustworthy state, and system recovery refers to the return of operational capability to the SCADA network as a whole.

In this research we will focus on the remediation process. Ideally remediation would eradicate the malware directly, without modification of other system files, such as most anti-virus products do. Unfortunately this approach assumes full knowledge of the malware and its infection. When victim to novel malware, the extent of the infection is not known. In this case a complete rebuild of the node is advised [13]. The most straightforward method of conducting such a rebuild is to restore the node from a trusted source image. Therefore, the remediation of a node involves the restoration of a node to a trusted state, and any actions associated with the detection or removal of malware. Consequently a recovery framework is one which permits the remediation of a number of nodes in parallel. A recovery framework can then be said to support system recovery.

2.1.1 SCADA Recovery Framework High-Level Requirements

In order to examine the SCADA Recovery Problem, a generic model of a SCADA system is developed. Our model of the system is based on the characteristics of SCADA networks we deem to be critical to the design of a recovery framework. While it is accepted that these characteristics may not be present in all SCADA networks we believe they apply to sufficient networks to make this research relevant. The key characteristics of a SCADA network are: a high degree of differentiation between nodes, lack of access to nodes, low forensics and response capability within the network operators, and steep availability demands.

There is a high degree of differentiation between nodes at both the software and hardware level in SCADA networks. This is in contrast to many data centres and administrative networks where workstations and servers will to a large extent run common images on identical hardware [4]. With the exception of redundant systems, the role played by each node in a SCADA system is unique enough to warrant both different software and configuration. The physical context in which the systems are installed often results in a different hardware composition, which in many cases results in a different software configuration. This means that the recovery framework will be required to maintain node specific configuration and software. Further to this, the framework should provide the network operators the ability to update the trusted software source for each node in order to account for changes to the network which must be made in situ [15].

The second key obstacle is the low level access to SCADA network nodes. This low level of access is observed both spatially and physically. Spatially speaking, nodes in a SCADA system are often located in close proximity to machinery or operator positions; this adds the overhead of a technician travelling to an infected node to the recovery timeline. When the availability constraints placed on SCADA systems are taken into account this overhead can become significant, especially when compounded over multiple infected nodes. In the physical sense, often nodes in a SCADA system are ruggedized to withstand the environment in which they are employed. The ruggedization process can restrict the ability a technician to access the computer hardware directly and enact a restoration. Sometimes security measures put in place to protect the hardware from intrusion must be removed to allow the technician access to the machine. These factors again work to delay the restoration of SCADA systems to a trusted operational state following an intrusion. As a result, the ability to actuate the recovery in both an automated and remote fashion is key to meeting operational timelines for recovery.

As mentioned previously on-site forensics and computer incident response capabilities are generally not expected to be adequate to conduct the necessary investigation and remediation activities. As a result the process is required to be entirely automated from start to finish. This eliminates the risk of human error undermining trust in a node following recovery. The recovery framework is also required to address malware persistence at the lowest level possible. This is because the lack of computer forensic expertise means that the full extent of the compromise will not be immediately ascertained. Consequently, the system cannot be trusted fully following the recovery unless the framework assumes the worst case scenario.

The importance of availability over integrity also requires the framework to prioritize the restoration of a node to an operable state over any other concerns, including guarantees of integrity. This means that, when possible, failure of the framework should leave a node in an operational state. As a corollary, the framework should indicate to the operator as succinctly as possible when a failure in an integrity measurement has occurred with enough information for operators to make a practical security decision.

These identified characteristics of a SCADA system lead to the following requirements:

1. The framework must preserve system specific software and configuration.
2. The framework must be fully automated and externally driven.
3. The framework must address malware at the lowest level possible.
4. The framework must prioritize availability over integrity.

A recovery framework for any SCADA network which adheres to the key characteristics described above must satisfy these requirements. Later in this chapter, further requirements which take into account an adversary's threat model will be added which specify a *trusted* recovery framework.

We argue that the requirements for automation and remote actuation are best served by a network based recovery scheme. A network based recovery scheme involves a centralized server which can use existing network infrastructure to conduct the recovery of each node. This allows the process to be scripted by the central recovery server and removes the need for a technician to physically visit each node. Nodes can use the recovery server's network boot services to boot into a remediation environment and be automatically restored to an operational state. With availability concerns addressed by making the recovery scheme network based our attention is given to the concern of integrity. The integrity of the a system following recovery can be tied to the integrity of the recovery process itself. The integrity of the process is threatened by the susceptibility of the recovery process to the malware it attempts to address and the extent to which the malware can survive the remediation procedures. Those topics are discussed in the following two subsections.

2.1.2 Attacks on the Network Recovery Process

The first threat to a network based recovery process we will examine is the threat of direct manipulation from an attacker. In this work, we assume a network based recovery to imply the use of some implementation of the Preboot Execution Environment (PXE) specification[16]. This assumption is based on the fact that as many SCADA networks adopt common IT practices, workstations are increasingly being built on standard computer systems [4]. Support for PXE is a common feature in Network Interface Card (NIC) firmware and PXE is an industry standard method of booting a machine from the network. A recovery process which incorporates PXE would involve configuring a target machine to boot into a remediation environment provided by a network boot server. Once in the remediation environment, the target machine can be restored to a trusted state.

In order to understand how a network recovery process is susceptible to attack the PXE network boot process will be briefly described. To start the process, the install target system's Basic Input/Output System (BIOS) is configured to boot from the NIC firmware. The PXE specification is implemented in the NIC firmware and makes use of the Dynamic Host Configuration Protocol (DHCP) and Trivial File Transfer Protocol (TFTP). The PXE process uses the DHCP protocol to configure the target's networking and to pass parameters necessary for further file transfer. These parameters include the TFTP server details and the location of a Network Bootstrap Program (NBP) on the server. The target will then retrieve the NBP from the TFTP server and load it into memory. The NBP is essentially a bootloader and it uses TFTP to download additional files in order to set up an initial boot environment. The initial boot environment implements further protocols such as the Hypertext Transfer Protocol (HTTP) or the Network File System (NFS) protocol which can then be used to download an image, such as the Windows Preinstallation Environment (WinPE) or a Linux kernel and initial RAM disk. These are then loaded into memory and executed, providing the remediation environment. WinPE is a lightweight version of the Windows Operating System which provides an environment with specialized tools for deploying, repairing, and configuring Windows installations. However, a Linux kernel and its root file system are more easily customizable to meet a network's needs.

Once the install target has finished booting into the remediation environment, the restoration of the node is begun. While there are a variety of potential methods which could be employed to recover a system we will consider the simplest, whereby a trusted image is retrieved from the network and restored to the disk. Once the system is restored it can be rebooted into an operational state.

The term BIOS is used above and in the rest of this paper to mean both legacy BIOS as well as a BIOS which implements the Unified Extensible Firmware Interface (UEFI) specification. Although UEFI is a drastic change from legacy BIOS it provides support for many of the legacy BIOS features[17]. From its outset, UEFI supported the legacy PXE boot process. Following version 2.5 of the specification, UEFI has provided advanced network booting capabilities natively, such as the ability to retrieve the NBP using FTP(S) or HTTP(S). Despite these improvements, the network booting process has fundamentally stayed the same.

Each stage of the above described process can be manipulated by a malicious actor. As mentioned, in the initial stages of a network boot the target system broadcasts a DHCP Discovery packet in order to initiate a DHCP session during which it will receive networking parameters along with the path to the NBP on the TFTP server. A malicious DHCP server on the network could hijack the process and give the attacker access to the system prior to OS-resident security protections[18]. The TFTP protocol also implements no security protections. Since the boot image is sent unencrypted a malicious actor on the network could inject packets to hijack the communication or corrupt the NBP[3]. In addition to this there are concerns depending on how the new disk image to be restored is transferred to the target system. If it is passed in the clear and uncompressed, any malicious actor on the network can observe the image in transit. This means that sensitive information, such as credentials, which are contained within the image are now known by the attacker. A malicious actor could also initiate the recovery process from a node they control on the network in such a way as to gain access to the image. These issues can be partially addressed by isolating the target system and DHCP / TFTP server from other network nodes using VLANs. An attacker would then have to accomplish a VLAN hop in order to subvert the process. If the network switch has been compromised, is not configured correctly, or not updated regularly, the threat of VLAN hopping is increased[18].

Since the network recovery process itself can be manipulated by malicious actors on the network, a target system restored as a result cannot be immediately trusted. We further state that even if the restoration process proceeds as expected, given current bootkit capabilities, there is no guarantee that the restored system is free from malware presence. The next subsection gives a brief overview of current rootkit / bootkit techniques and the impacts on a potential recovery process.

2.1.3 Methods of Malware Persistence

A central concern of the recovery process is how concretely it addresses the problem of persistent malware. Malware is difficult to detect and remove by design. As mentioned previously, given that malware or its malicious effects are detected, a comprehensive forensics review of the system is commonly outside of the capability of SCADA system operators. Such an investigation would interrupt system operation and fails to satisfy the availability constraints. This means that the extent to which the malware has compromised an individual system or spread on the network will not be immediately known. As a result of this, operators must assume that the malware will attempt to install a root or boot kit, and actions should be taken accordingly.

A rootkit is a software tool used to covertly maintain persistent presence on a system. While many rootkits are installed within the context of the operating system, some utilize kernel-mode privileges in order to install themselves to system boot sectors or at even lower levels such as the system firmware. In these cases the term 'bootkit' is used to refer to the malware. The forthcoming book *Rootkits and Bootkits* provides an excellent description of the field's recent history[19]. The recent history has largely been colored by the emergence of Microsoft's Kernel-Mode Code Signing Requirements with Windows Vista 64-bit [20]. This policy prevents Windows from loading any kernel-mode driver which has not been signed by Microsoft. This disrupted one of the most common means of infecting the Windows boot process. As a result, one of the ways in which current bootkits can be classified is the technique they use to circumvent kernel-mode code signing policy. *Rootkits and Bootkits* identifies three circumvention techniques employed by bootkits: User Mode, Kernel Mode, and System Firmware. User mode bootkits exploit the methods provided by the operating system for turning off the integrity checks; subsequently their persistence does not outlive a reinstallation of the operating system. Kernel mode bootkits attack kernel-mode memory during the boot process. They reside in the Master Boot Record (MBR) or Volume Boot Record (VBR) and therefore do not survive the re-imaging of the system disk, consequently US-CERT recommends that a system should be formatted and rebuilt following detection of a bootkit [21]. Finally, firmware bootkits reside in the firmware and attack the system boot process directly. This type of bootkit is the hardest to both detect and address. Depending on which firmware specifically is targeted, response actions range from flashing firmware to outright replacement of hardware. In addition to proof of concepts developed in academic settings, firmware bootkits for both BIOS and UEFI have been detected *in-the-wild*. The first observed *in-the-wild* BIOS bootkit, `Mebromi`, was detected in China in 2011 [22]. The 2015 Hacking Team breach revealed a UEFI bootkit targeting Insyde BIOS [23], the most widely used BIOS [24].

The previous two subsections have described how the ability to trust a system following a network based recovery approach is undermined by threats to the network boot process and the persistence techniques used by modern bootkits. The next section presents a threat model which specifies how an attacker would use the above techniques to compromise a SCADA network.

2.2 Threat Model

In our threat model, we permit the attacker limited physical access to the network. We further qualify limited as enough access to compromise a node locally but not enough to interact directly with hardware. This means that side channel attacks against cryptographic hardware which require lengthy physical access, such as those revealed by The Intercept in [25], are outside of the adversary's scope. We assume that the attacker has the ability to obtain system level privilege on the compromised node and thus has the ability to install a bootkit in the system's boot sector. We grant the attacker further capability in that they may have the ability and intent to modify the boot firmware in order to install a bootkit or adjust the configuration to enable further compromise. As a result, the attacker has full control of the system following a compromise.

Further to this, once the adversary has a foothold on one node, we assume they intend to exploit further nodes on the network. We give them sufficient capability in order to be successful in these attempts and any systems with similar operating systems such as redundant nodes are also assumed to be compromised. Dissimilar nodes on the same network may be compromised as well, this includes network infrastructure such as switches and routers. Through access to these compromised nodes the attacker has control over the network. This control allows the attacker to monitor, modify, and inject packets into ongoing network communications.

The implications of this level of attacker access are threefold. The attacker can install kernel mode rootkits, boot sector bootkits, and firmware bootkits. The attacker can traverse the network laterally and gain equivalent access on multiple nodes. The attacker also has the ability to conduct replay or man-in-the middle attacks on the network.

These implications lead to two operating constraints for a *trusted* recovery process.

1. The framework must be able to operate without trusting network communications
2. The framework must be able to operate without trusting any software on the nodes

What is needed is a way to measure the recovery process itself such that the authenticity of a node can be verified following recovery given that neither the network nor the node itself can be relied upon. The following section provides a more formal definition of trust and describes how trust is bootstrapped in modern computing systems.

2.3 Bootstrapping Trust

A trusted system, in the context of computer security, is one which acts in an expected manner for an intended purpose[26]. In order to determine if a system will act in the expected manner for its intended purpose the state of the system must be verified. Parneo, et al[27], present a thematic summary of bootstrapping trust in modern computer systems. In general, a system's state can be determined by measuring the identity of the code running on it. Code identity is made up of the program binary along with applicable inputs, libraries, or configuration files. As a result, code identity is best measured by calculating a hash of the code and its properties.

In order to trust a code measurement the code identity of the software which conducts the measurement must be itself measured and verified. From this, follows the idea of integrity chaining. This concept was presented in 1997 by Arbaugh, et al in the course of proposing a bootstrap architecture, AEGIS [7]. Arbaugh, et al present an ideal chain of integrity checks used to sustain system integrity with the recurrence:

$$\begin{gathered} I_0 = True, \\ I_{i+1} = I_i \wedge V_i(L_{i+1}) \quad for \quad 0 < i \end{gathered} \tag{2.1}$$

The integrity at level i is represented by the boolean value I_i. Each level has a corresponding verification function, V_i which takes the next level, L_{i+1}, as an argument and returns true if the next level is successfully verified. The boolean `and` operation is depicted with $\wedge$, consequently, equation 2.1 formally states that the integrity of level $i+1$ is preserved provided that level i is trusted and the verification function over level $i+1$ succeeds.

As in AEGIS, this concept can be applied to the boot process of a computer system, where each *level* of the boot process (for example: the BIOS, bootloader, OS.. etc) measures the following level before execution. I_0, then, is the code block which conducts the first measurement and it must be trusted implicitly. This is referred to as Root of Trust for Measurement (RTM) or as the Core Root of Trust for Measurement (CRTM) as it provides the root of trust upon which all subsequent measurements are based. As a result the CRTM often resides early in the boot process and, ideally, is protected from manipulation by a user. The process of integrity chaining from a CRTM towards the runtime state of the system is referred to as *trusted* or *measured* boot. In situations where the CRTM is contained in the BIOS boot block it is often referred to as a Static Root of Trust for Measurement (SRTM).

The most commonly used measurement process is with the use of a cryptographic hash function. A hash function maps data of arbitrary length to a hash value of a fixed size. In order for a hash function to be considered cryptographic is must be both one-way and have a low collision rate. A hash function is one-way when it is easy to calculate the output hash and it is computationally in-feasible to calculate the input data given a hash value. A low collision rate means that users of the hash function can assume, with a reasonable probability, that two different inputs will not result in the same hash value. In trusted computing the SHA-1 and the SHA-2 family of hash functions are commonly used.

Since maintaining a long chain of trust is often difficult, and in order to provide a trusted environment after a system's boot has been completed, Intel and AMD have implemented instructions for a Dynamic Root of Trust for Measurement (DRTM) into their chipsets [28]. A DRTM allows a trusted execution environment to be created during runtime. The key benefit is that security critical code can be run in a measured environment without having to implicitly trust the BIOS, boot loader, or OS.

The following two subsections will explore the concepts of a SRTM and DRTM in more depth by describing the Trusted Platform Module (TPM) and the associated late-launch capabilities provided by Intel Trusted Execution Technology (TXT).

2.3.1 Trusted Platform Module

The most widely adopted implementation of a hardware root of trust is the TPM. Root of trust in this sense does not imply that the TPM implements a

RTM, as this function is provided by the BIOS, but that it implements a Root of Trust for Reporting (RTR) and Root of Trust for Storage (RTS). A TPM is any implementation which meets the interface and isolation requirements of the TPM specification developed by the Trusted Computing Group (TCG) [29]. The TPM is almost ubiquitous on enterprise hardware and also occupies a growing proportion of consumer computers, including Google Chromebooks. In 2014 the US Department of Defense mandated all new computer purchases, from desktops to servers to mobile devices to include a TPM [30].

The TPM Specification calls for a secure coprocessor capable of performing cryptographic functions, unalterable Platform Configuration Registers, and a unique endorsement key. A Platform Configuration Register (PCR) is used to store aggregate measurements of software components as they are loaded during the boot process. The endorsement key uniquely identifies the TPM and client system and is used to generate other keys used to sign attestations. The TPM provides two functions related to maintaining a chain of trust and providing attestation: *extend* and *quote*. *Extend* is the mechanism by which measurements are stored in a PCR. The TPM calculates a digest of the existing measurement concatenated with the new binary data to be measured and stores it in the PCR. For example: binary data, d, is extended into a PCR containing measurement m_i using hash function H with:

$$m_{i+1} := H(m_i||d) \tag{2.2}$$

The ability to securely store measurements in PCRs and the *extend* function provide the RTR as it allows the system state to be captured in a manner in which the result cannot be modified. *Quote* is the mechanism used to support attestation. The TPM takes a nonce from a verifying system and creates a signed statement including it and a composite hash of a set of PCRs. The statement is signed with an attestation identity key (AIK) which is derived from the endorsement key. Attestation involves a TPM quote along with the list of measurements taken by the TPM which resulted in the PCR's current state. The verifier then validates the attestation by both verifying the signature and comparing the resultant measurements to a list of golden measurements.

The RTS is provided by the ability of the TPM to encrypt data outside of the TPM Non Volatile Random Access Memory (NVRAM) using a secret stored within the TPM. This functionality is not used in this research and therefore will not be further expanded on.

The above capabilities can be utilized to support integrity chaining and trusted boot processes. Figure 2.1 depicts the trusted boot process as described for a UEFI BIOS. Following a PC reset, the boot rom verifies that an

Authenticated Code Module (ACM) has been signed by the vendor's private key [31]. This verification is the RTM for the system and is highlighted in the diagram with a red star. Once the ACM has been verified it is executed and, in addition to its own start up functionality, extends PCR zero with a measurement of an initial BIOS boot block before passing execution to the BIOS. From here subsequent modules and configurations of the BIOS are hashed and extended into PCRs zero through seven. Figure 2.1 depicts the allocation of UEFI measurements across these PCRs. Once the platform initialization procedure is complete and execution is passed on to an operating system a verifier can request a quote of the applicable PCRs and compare the quote to prerecorded *golden measurements* for the BIOS [8]. PCRs 8-15 are available for use by the operating system.

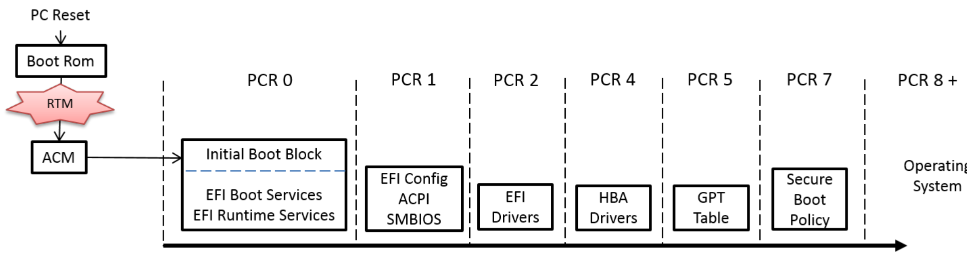

Figure 2.1: Trusted boot sequence with UEFI firwmare [1]

The measured boot process allows for changes in the BIOS to be detected. This is important in two scenarios. The first is to detect changes to the BIOS configuration. These changes could be the result of improper use by a network operator or the result of a malicious actor. In either case the BIOS configuration has deviated from an established configuration and the system may now be more exposed to attack. It is also important to detect when the BIOS has been modified by a bootkit and some malicious process is started as part of the platform initialization process. Detection of these types of changes to the BIOS is complicated by the fact that the SRTM could also be modified by the adversary, which compromises the entire chain of trust.

Drawbacks of a Static Root of Trust for Measurement

Since the RTR is implemented in the TPM it provides a relatively strong guarantee of integrity. The SRTM, as mentioned, is implemented in firmware by the BIOS vendor. Since the integrity of all subsequent measurements is based on the extent to which the SRTM can be trusted, this is an area of some concern. In fact there are several issues with the use of a BIOS SRTM as a RTM.

The first concern is related to the fact that a trusted boot process is only trustworthy if the SRTM itself is trustworthy and cannot be modified [32]. Researchers have shown that the previous assertion to not be true for common BIOS implementations [32, 33]. As a result, proof-of-concept malware has been developed which modifies the BIOS firmware and provides false measurements to the TPM in order to fake authorized integrity measurements [33]. Another criteria for trusting the trusted boot process is that the hash chain be contiguous, that is that the integrity measurements have full code coverage and no code is executed without being first measured. This condition has also been noted to not always be the case in BIOS implementations [32].

The first concern noted above is addressed by Intel's Boot Guard feature, in which the boot ROM is stored in One-Time-Programmable memory. The boot ROM then verifies an ACM using a key loaded into field programmable fuses during manufacture. The ACM is then relied upon to measure an initial boot block in the BIOS which begins the integrity chain[31]. These details have been included in Figure 2.1. When available, Intel Boot Guard does improve the situation but it does not address the reliance on the vendor to adequately cover their BIOS with integrity measurements. Further to this, maintaining a hash chain through the BIOS, a bootloader, and an operating system boot manager is an onerous task to implement properly. For this reason modern Central Processing Units (CPUs) now provide a *late-launch* capability in order to allow a measured launch environment to be created from a DRTM. The Intel implementation is referred to as Trusted Execution Technology.

2.3.2 Intel Trusted Execution Technology

Intel Trusted Execution Technology (TXT) is Intel's implementation of a DRTM [2]. As mentioned previously, a DRTM is a root of trust which does not rely on a chain of trust from a SRTM. This is achieved by conducting a 'late-launch', effectively a reset during runtime, in order to reset the platform to a known state, measure a piece of code and provide it a hardware protected environment in which to execute.

Similar to as how the static trusted boot procedure was depicted in Figure 2.1, an illustration of the TXT launch sequence is provided in Figure 2.2. There are two principle components that make up the TXT launch sequence: A Secure Init (SINIT) ACM and a Measured Launch Environment (MLE). The SINIT ACM is, much like the ACM used during modern boot processes, a chipset specific code module which the vendor has digital signed. The MLE is a code module associated with the trusted operating system being launched such as bootloader or virtual machine manager. Prior to commencing the TXT launch sequence the SINIT ACM and MLE are loaded into memory. Additionally, any further modules required for the trusted environment are also loaded into memory [2]. In the example presented in Figure 2.2 a linux kernel and initial RAM disk are also included in the sequence.

The launch process begins with the GETSEC[SENTER] instruction. This instruction causes all but a system bootstrap processor to perform clean up and enter a wait status. It also resets PCRs 17 to 22 in the TPM. The microcode associated with the GETSEC instruction is the only entity authorized to reset PCRs outside of system boot. Once these steps have been completed the SINIT ACM is measured, verified, and executed. As shown, the measurement is extended into PCR 17. The SINIT ACM confirms that the chipset and processor have a suitable configuration before measuring and launching the MLE. This measurement is stored in PCR 18. In some cases, such as when the `tboot` bootloader is used, a Launch Control Policy (LCP) which describes how the MLE should be established is also measured and extended into PCR 18 [34]. At this stage the MLE redirects interrupts, uses Intel Virtalization Technology for Directed I/O to implement protections from other devices with Direct Memory Access (DMA), and sends wake up signals to the waiting processors. It will also implement the LCP. The LCP can specify valid measurements for PCRs 0-7 as well as for the MLE and any additional modules. If these valid measurements are not found during the launch process it can either fail to launch or halt the late-launch process and launch normally.

Once this stage in the sequence has been reached, the MLE will measure and store hash values for any additional modules, such as the kernel and initial RAM disk shown in the figure. The final stage of the launch sequence in this case would be for the MLE to pass execution to the kernel, which will run in the newly created trusted environment.

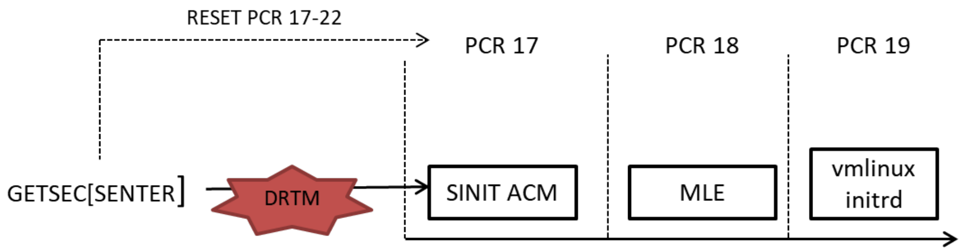

Figure 2.2: Intel TXT late-launch sequence [2]

As can be seen, Intel TXT provides a way for a trusted environment to be created from an untrusted system. Since the processors are started directly into protected mode with the MLE, as opposed to into real, virtual, and then protected mode as in the normal bootstrap process and the trusted OS is protected from devices with DMA, such as exploited NIC firmware, the environment following TXT can be said to be more deterministic than the OS environment following a standard trusted boot[35].

That is not to say that the environment is totally isolated. Both System Management Mode (SMM) processes and the Intel Management Engine (ME) can both access memory in the trusted environment[36][28].

In this section the idea of trust in computing systems has been explored. The existence and validity of various roots of trust has been discussed and common implementations were described. These are relevant to the design of a trusted recovery framework as they provide a means through which the recovery process can be measured and tied to a root of trust in hardware. The next section will continue to explore advanced chipset features in computing hardware with the Intel Management Engine, a remote administration capability implemented in hardware. These remote administration capabilities aid the design of a trusted recovery framework as they provide the ability to remotely control a target system outside of the context of an operating system.

2.4 Intel Management Engine

Intel's ME was introduced in 2007 as a platform in which to implement Intel's Active Management Technology (AMT), which at the time was only resident on certain network cards[37]. Since then it has grown to include more modules, all of which are collectively marketed as 'Intel vPro'. At its core the Intel ME is a dedicated microcontroller contained in the Chipset Memory Controller Hub along with dedicated storage and memory. While previous versions ran a proprietary Real Time Operation System (RTOS) on Argonaut RISC Core (ARC) architecture, as of Version 11 it runs x86-based MINIX 3 [38]. Aside from level 1 and 2 caches located within the ME microprocessor it uses 16 MB of system memory from DIMM slot 0 for paging and shares the SPI flash NVRAM on the motherboard with the BIOS and network controller for firmware storage[37]. It is resident on nearly all modern x86 based computers. Its level of system access and closed source nature has led to numerous security concerns[36][39][38].

While originally designed as a remote administration feature, Intel ME now implements several security functions including housing an integrated TPM and supporting real time media decryption for Digital Rights Management[31]. However, its flagship feature is Active Management Technology (AMT).

Intel Active Management Technology

AMT is a feature designed to give systems administrators a suite of remote administration capabilities. Because it is implemented on the ME it can provide functionality that would only otherwise be possible when physically at the system. This functionality is provided 'out-of-band', meaning that communication with the ME happens outside of the local operating system and its network stack. As result of this, a systems administrator is able to interact with a system even when the OS network connectivity is corrupted[37]. It also allows an administrator to change BIOS boot options and reset the computer; one use case of this feature would be to configure the system to conduct a PXE boot in order to redeploy the operating system from a net boot server. Another unique feature is IDE-Redirection (IDE-R). Using IDE-R, an administrator can configure a target system to boot from an image file stored on the admin system as if it were local media over an encrypted serial connection. It can also act as a hardware firewall, intercepting packets between the NIC and the operating system. AMT can communicate with the host OS through a Host Embedded Controller Interface which allows for application monitoring and persistence features.

Communication with the Intel AMT over the network is secured using Transport Layer Security (TLS) and authentication is provided by HTTP digest or Kerberos. Administrators can either issue commands through a web server hosted on the AMT or by utilizing management applications which make use of Web Services-Management (WSMAN), Keyboard-Video-Mouse (KVM) over network, or serial over LAN protocols[31].

2.5 Related Work

In the previous two sections the idea of trust in computing systems was developed and technologies which allow trust to be bootstrapped from protected roots of trust were explored. This section will discuss implementations of these technologies in similar problem areas.

Schiffman et al[3], suggest a network Root of Trust for Installation (ROTI) process. A ROTI is an operating system installation which is conducted in such a way as to be trusted implicitly. Subsequent to the installation, the current state of system files can be measured and then compared to the ROTI in order to detect tampering. In order to create this root of trust installation late-launch technology is used to create a trusted environment on a target machine following a PXE network boot. In this case AMD's equivalent to Intel TXT, Secure Virtual Machine (SVM), is used to create a DRTM. The DRTM is used to measure the installer which then installs and configures Linux on the host, creates a record of the filesystem, and signs it using the TPM. This operation is given by:

$$P = Sign(F, D, I)_k \tag{2.3}$$

That is, a ROTI proof P, can be created with measurements of a recovery installer I, using disk image D, and resulting in file system F signed by k, a key unique to the target's TPM. This signed record is a Root of Trust for Installation which can be leveraged later to attest to platform integrity. Figure 2.3 provides a visual overview of how the netROTI was implemented. Steps 1-3 represent the PXE network booting process. Step 4 is the *late-launch* procedure being executed by the oslo bootloader, a bootloader which used AMD's SVM to create a DRTM. Step 5 depicts the storage of the DRTM in PCR 17. Following this some initialization code is executed in step 6 and it in turn measures any of the modules supplied to it. In this case it measures the Linux Kernel and initial RAM disk as in step 7. Following step 7, the installer proceeds to build the system and generate the ROTI proof.

This work is important to the design of a SCADA recovery framework because it demonstrates how PXE can be used to chain load into Intel TXT (or its equivalent) and create a DRTM. This means PXE and Intel TXT can be used to create a measured environment in which remediation activities can be conducted and whose trust isn't anchored in the potentially compromised BIOS. It is also valuable because the ROTI proof demonstrates how the DRTM can be leveraged to generate measurements which cover the entirety of the remediation process. The boot chain from the DRTM to the trusted operating system is stored in the TPM PCRs following the late-launch process. The ROTI proof includes these measurements along with the measurement of additional inputs (the disk image) and the result (the filesystem). This proof then can be used to establish both the integrity of the installation process and, through the TPM signature, the identity of the node.

Unfortunately this work is not directly applicable to the SCADA recovery problem. Principally, SCADA systems are characterized by their highly differentiated and hierarchical nature and this approach does not accommodate this. Furthermore, while this approach protects against adversarial interference with the PXE boot process, it does not address the issue of malware presence in firmware directly.

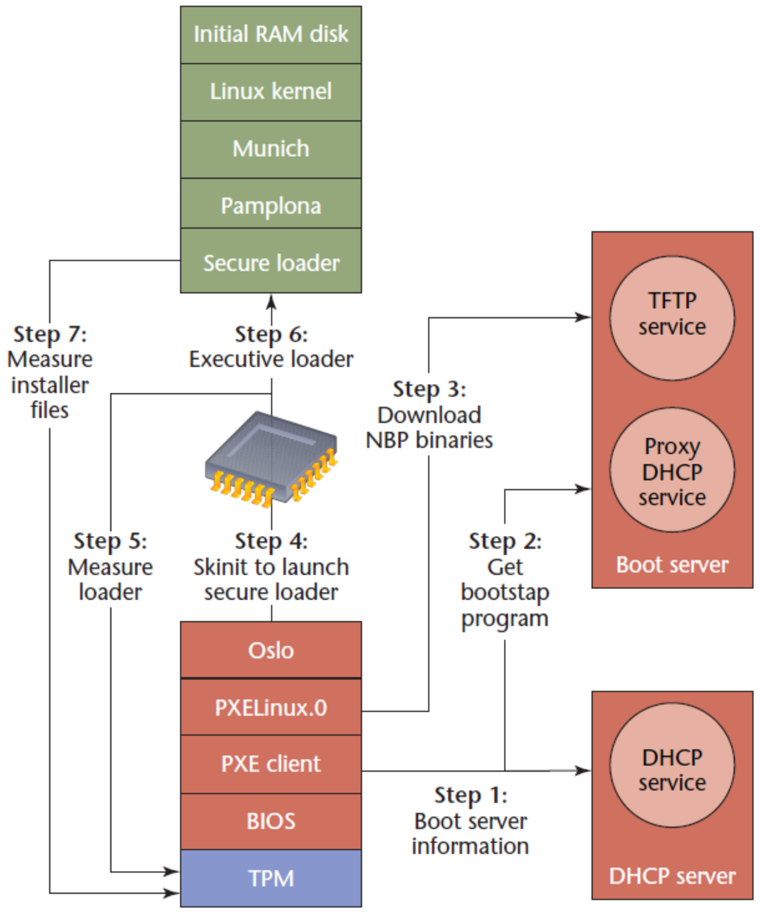

Figure 2.3: Network root of trust for tnstallation [3]

The use of Intel TXT in cloud environments has also been seen in practice with both the OpenCIT and crowbar projects [40][41]. These are both deployment software designed to work as part of OpenStack. OpenStack provides a framework for the management of cloud computing pools and OpenCIT and crowbar deal with the issue of provisioning a bare-metal server with a hypervisor over the network in a trusted manner. Due to the differing nature of cloud computing networks and SCADA networks it is inappropriate to adapt one of these software and the OpenStack architecture to serve the needs of a SCADA system. However, cloud computing networks do share a key characteristic with SCADA networks, the geographic displacement of computer systems. In cloud environments this is dealt with by employing remote administration tools such as Intel AMT, the Intelligent Platform Management Interface (IPMI), or other technologies which utilize a system's Baseboard Management Controller(BMC) to remotely control a system. These capabilities should also be utilized in a recovery framework for SCADA systems to address the issue of low access to nodes.

In this chapter SCADA systems and the problem of recovering them following a computer intrusion were discussed. The potential of a network remediation process to address malware was presented, followed by a description of ways in which an adversary could undermine our trust in such a process. Hardware security features which can be leveraged to bootstrap trust despite adversarial activities were then described. This was followed by descriptions of implementations of these technologies which illustrate how they might serve the SCADA recovery problem.

3 Design

In the first chapter, we described the need for a trusted recovery architecture for SCADA networks. In the second chapter we elaborate on the problem of recovering SCADA systems in a trustworthy manner, examined more formally what is meant by trust in computing systems, and looked at hardware mechanisms which can be leveraged to establish trust in computing systems. As described, none of the software which implement those hardware mechanisms adequately meets the need of a SCADA recovery framework. The task in this chapter is to describe the design of a trusted recovery architecture for a SCADA system. This is done by first enumerating the requirements of the recovery framework and then describing an architecture which meets those requirements. The resultant architecture is intended to be simple, extensible, and reliable.

3.1 Design Considerations

The design of the framework is driven by our model of the system to be recovered, the threat model, the trust relationships we wish to generate, and assumptions about how the framework will be employed in practice. The system model encompasses the characteristics of a SCADA network which are most typical of those found in use and the attacker model describes the capabilities of a notional adversary on the SCADA network. These two models were presented in the previous chapter, Sections 2.1 and 2.2 respectively. As a result of these two models a set of high-level requirements for a SCADA Trusted Recovery framework were developed. These requirements were used to guide the design of the framework described in this chapter. In addition to the high-level requirements, a trust model and some assumptions were considered. The trust model results from the threat model as well as the roots of trust available on the SCADA platforms. The trust model lays out which aspects of the network we are able to trust and what will be deemed untrustworthy.

The design will then seek to leverage the trusted aspects of the network in order to implement a trusted recovery procedure. The assumptions describe the manner in which we expect the recovery framework to be deployed.

3.1.1 Trust Model

The threat model in 2.2 identified which aspects of the system we cannot trust. It is now important to identify what can be trusted and what has to be trusted in order to recover the system to a known good state. While we have stated that the attacker has full access to a compromised node and we cannot trust any software on the node, we have excluded hardware attacks against the computer hardware. This eliminates the attacker's ability to conduct attacks against the TPM such as those discussed in [32], [42], and [25]. Consequently we decide to trust the TPM as a Root of Trust for Reporting.

Along with the TPM we also decide to trust the DRTM created by Intel TXT or AMD SVM. This is because key aspects of late-launch capabilities are implemented in CPU micro-code associated directly with the late-launch CPU instructions. Specifically, the verification of the vendor signed Authenticated Code Module and the reset of the late-launch specific PCRs can only be done using specific privileges levels obtained by the CPU during the process [2]. This creates the DRTM upon which the remainder of the late-launch process is based.

Since we trust the TPM's RTR and Intel TXT's DRTM we have roots of trust for both measurement and reporting. Subsequently it is possible to create a hash chain from the DRTM into our remediation environment. It is also possible for the remediation environment to conduct remote attestation and provide its platform state to a verifier. These two capabilities are key to the design of a trusted recovery process. Since the remote attestation process occurs over the network, and we do not trust the network, the claim that we can conduct remote attestation may be questioned. However, while it is true that an attacker with control of the network could disrupt network communications and cause a denial of service, due to a lack of access to the TPM's private key they do not have the ability to forge integrity measurements and fool the verifier about the state of a node following recovery. Additionally, the inclusion of a psuedo-random nonce in the TPM quote process prevents replay attacks.

Leveraging the CPU late-launch capability is also what the netROTI process detailed in Section 2.5 was built upon [3]. It is also suggested by NIST as a potential model for BIOS remediation [8]. It should also be noted that the Threat Model described in 2.2 is similar to that which the Trusted Computing

Group seeks to address with the TPM[29]. It is also similar to that used by Maene, et al, in their comparison of 12 hardware-based trust architectures [43]. These similarities are valuable because they allow the effectiveness of the design to be compared to the guarantees provided by the underlying technology. That is, if the recovery framework cannot provide the same potential integrity performance as the TPM or the same functionality as that identified by Maene, et al, then a sufficient reason should be clearly identified.

3.1.2 Assumptions

By focusing on the ability to simultaneously remediate a set of nodes to a known good state, this framework is designed to be utilized as part of a broader recovery plan. As such, several assumptions about this broader network recovery plan and the nature in which the framework will be employed need to be made in order to conduct the design. The wider response plan is assumed to implement a tiered approach in which, perhaps, the network is subdivided into groups based on functionality and these subgroups are recovered in turn. This implies some method, physical or virtual, of segregating the network. This then leads to the important assumption that the segregation policy protects newly remediated systems from yet to be remediated, still compromised, systems. It is also assumed that the nodes which are to be remediated can be safely shutdown without causing an unacceptable loss of service or safety.

Verification of platform state is conducted based on comparison to previously stored, golden measurements. Therefore, in discussions of the verification process it is assumed that these golden measurements were captured from non-compromised systems. Furthermore, anytime system state is captured, such as when images are cloned from a node, the node is again assumed to be in a non-compromised state. This means that the golden measurements and repository of authorized images are trusted. Another way to state this assumption is that we only assume an adversarial environment during the restore process and not any supplementary processes required to support the recovery framework.

Finally, we assume the server and any accompanying image repositories used to conduct the recovery can be adequately protected. In addition to the TPM and Intel firmware, these must be trusted implicitly in order to trust the recovery process.

3.2 Design

The high level requirements from Chapter 2 are summarized in Table 3.1.

Table 3.1: High level design requirements for a SCADA trusted recovery framework

No.	Requirement
1.1	Restoration must maintain system specific configuration and software
1.2	Framework must be fully automated
1.3	Framework must address persistent malware
1.4	Framework should fail to an operation state
2.1	Framework must not trust network
2.2	Framework must not trust node software

In the remainder of this chapter a design for a trusted recovery architecture which meets all of the requirements in table 3.1 will be presented. First the high level architecture of the framework will be described. Following this pertinent details of the architecture will be described in greater depth. Once the framework has been presented the system operation procedures will be outlined in order to illustrate its use.

3.2.1 High Level Architecture

The Trusted Recovery Architecture is designed to address the system constraints listed in the previous section. It incorporates CPU-based late-launch and out-of-band remote administration technology in order to address the threat of an adversary as described in our attacker model. It is designed to be modular and extensible such that it can accommodate new hardware with differing capabilities. The major components are split between the node being recovered and a Recovery Server which facilitates the operation. There are four operations the framework can undertake on a node. While these will be discusses in greater detail in a later section, they are described here, briefly:

- **Initialize** Take ownership of a node's TPM, generate an AIK pair and store the public key and Universally Unique Identifier (UUID) on the Recovery Server. This is used when adding new hardware to the system.
- **Measure** Generate golden measurements for the BIOS and a Trust-Stage Bootloader and store them on the Recovery Server.
- **Clone** Clone a node's disk, calculate a hash value for each image cloned, and store the image and measurement hash on the Recovery Server.
- **Restore** Restore a node to a previously saved image. As the restore process proceeds, measurements of all aspects of the process, including the BIOS, Trust-stage Bootloader, and disk images are stored in the

TPM. When the process completes these measurements are remotely attested to an external verifier, the Recovery Server.

The remote attestation process at the end of the restore operation is key to developing trust in the recovery process. In this framework we refer to the remote attestation process as the verification phase. The first three operations establish the node identity and generate golden measurements which represent the trusted system state. The restore operation allows a system to be verifiably restored to the determined trusted system state. These operations are supported by the high level architecture as depicted in Figure 3.1.

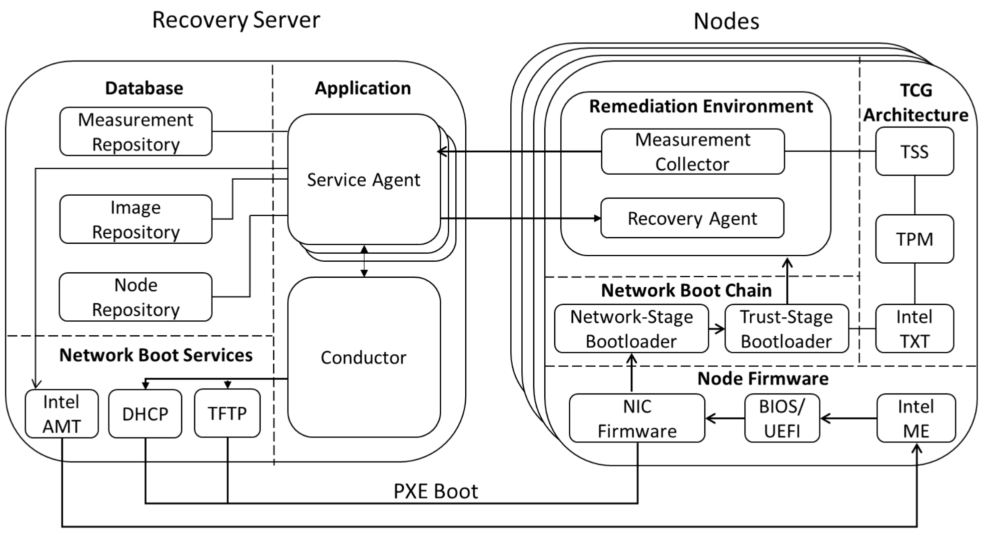

Figure 3.1: Trusted recovery framework architecture

In general, the Recovery Server conducts a remediation or associated action on a set of Nodes by leveraging features in the Node's hardware and firmware. These features include an out-of-band remote administration capability, such as Intel AMT to remotely initiate a PXE boot, the NIC firmware to boot the Node from a Server provided image, and the TPM to support remote attestation. The high level components on both the nodes and on the server are described in the following list:

- Nodes:
  - **Network Boot Chain** The network boot chain is composed of a network-stage bootloader responsible for chain loading the trust-stage bootloader and remediation environment and the trust-stage

bootloader itself. The trust-stage bootloader is responsible for initiating the DRTM and establishing the Remediation Environment.

  - **Remediation Environment** This is the environment from which all remediation tasks are conducted. These tasks include the cloning and restoration of images along with the capture of integrity measurements. It consists of a recovery agent which interacts with the system disk in order to clone or restore system images. It also contains a measurement collector which is responsible for reporting integrity measurements to the Recovery Server.

- Recovery Server:
  - **Network Boot Services** These services include remote administration functionality for remotely configuring nodes to conduct a PXE boot, DHCP services for assigning DHCP leases to nodes and directing them to a TFTP server, and TFTP services to serve the Network Boot Chain modules to the node.
  - **Database Layer** The Database Layer provides repositories of data for the nodes, their authorized images, and golden measurements to be compared against. Node data includes the data required to support the establishment of node identity. Image data includes a path to the image resource, which may or may not reside on the Recovery Server. Measurement data includes the trusted measurement values for the BIOS/UEFI firmware, the trusted images, and the Trust-stage Bootloader and Remediation Environment.
  - **Service Agent** The Service Agent is responsible for communicating with the remediation environment on a node in order to support its ongoing operation. This support includes receiving data from the node to be stored in the database, providing authorized images, and receiving TPM quote data. TPM quote data is compared to authorized values stored in the database and used to validate both the signature and signed hash in order to establish both node identity and integrity. The Service Agent is specific to the type of node, therefore a different one must be implemented for each type of node. Additionally, during execution, a separate Service Agent object must be instantiated for each node being operated on.
  - **Conductor** The conductor coordinates actions across multiple nodes. This coordination includes the commencement of network boot services and creation and execution of a Service Agent for each node being remediated. It is also responsible for receiving events from the service agents, logging data, and reporting the status of integrity measurements from each node.

These components will be described in greater deal in the following two sections. Following these lower level descriptions the framework operation will be described in order to illustrate how the framework satisfies the requirements we have laid out.

3.2.2 Recovery Server

The Recovery Server is made up of a Network Boot Services Layer, which provides network boot functionality for nodes, a Database layer, which provides repositories for node data, and an Application layer, which consists of the operational logic of the framework.

The Network Boot Service Layer provides DHCP and TFTP services to the network and provides out-of-band remote administration functionality to the application layer using Intel AMT. The DHCP and TFTP servers are required to support the PXE network boot process. The DHCP server is configured to statically assign IP addresses to nodes based on MAC address. This allows the framework to establish node identity at the earliest stage and address node uniqueness throughout the process. The TFTP server allows the node to retrieve the Network-stage Bootloader, Trust-stage Bootloader, and the Remediation Environment. These services should be enabled during operation of the framework and disabled when the operation is completed. This limits a potential adversary's ability to gain information about the remediation process when it is not being conducted. Out-of-band remote administration functionality is required to remotely configure the node BIOS to conduct a PXE boot on the next power cycle. This functionality is available in Intel's AMT and can be be achieved through the Simple Object Access Protocol (SOAP) interface to the Intel Management Engine. Authentication is accomplished using a username and password stored for each node in the data layer. AMT functionality is provided to the Application layer so that a node's service agent can conduct the BIOS configuration and system restart as part of its remediation process. These capabilities allow for the initial stages of each system operation to be automated. By automating the process of initiating an operation on a node the Boot Service Layer partially fulfills Requirement 1.2 from 3.1, that the Framework be fully automated.

The Database Layer provides an interface to a database so the application layer can store and retrieve data during the framework's operation. The Data Layer implements repositories for the different data required by the framework. As previously stated this data includes node data, image data, and measurement data. Node data includes MAC address, assigned IP address, AMT credentials, TPM key data, and node type. Image data concerns data relevant to a particular restore image, such as the date it was cloned, the network path to download it, and measurement hashes for each image. Measurement data includes measurement hashes for the remediation environment and boot chain. Measurement data is also specific to the type of node as opposed to specific to a particular node, as with the image hashes.

Finally, the Application layer implements the operational logic of the Recovery Server. Its design is inspired by the Command Design Pattern defined in [44]. The pattern has been modified with the addition of a conductor, which is composed of a list of command objects and is responsible for executing each command in its own thread. The Class Diagram in Figure 3.2 illustrates the Recovery Server architecture.

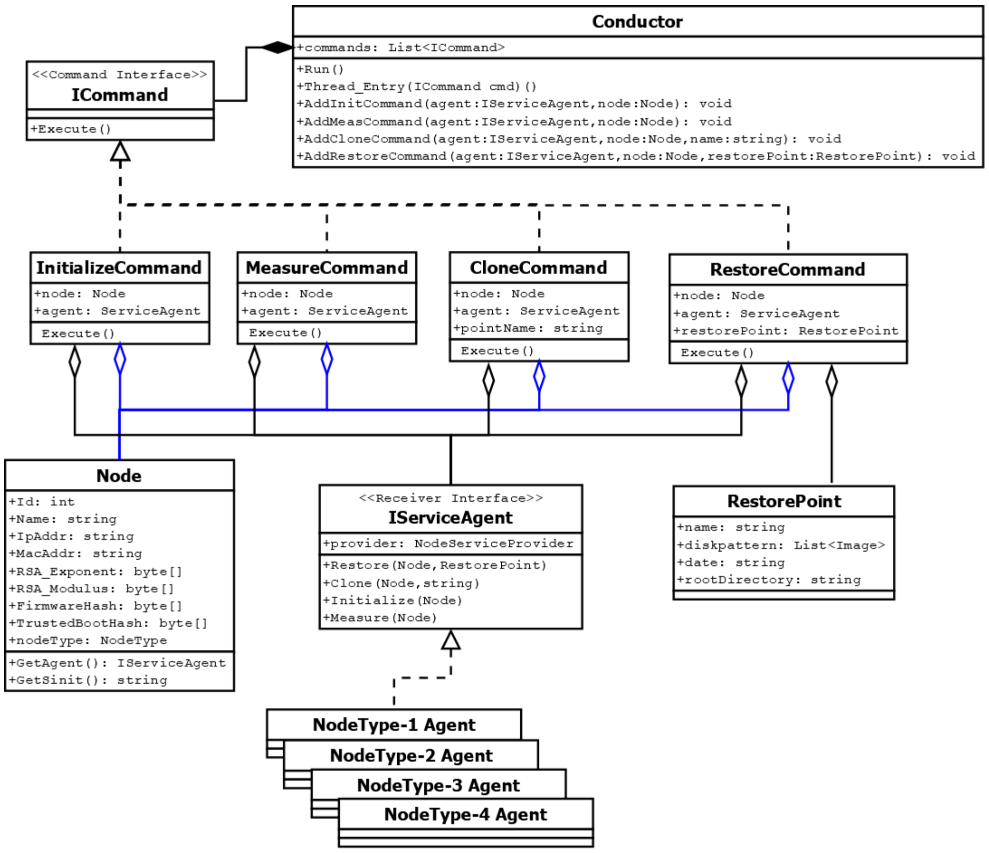

Figure 3.2: Class diagram of the recovery server architecture

As shown in Figure 3.2, a Conductor object is composed of a list of commands. It includes methods which allow various command types to be added to this list. It also provides a `Run()` method which executes each command. As in the Command Design Pattern [44], the Command interface and its implementations provide decoupling between the calling object and the actual task logic. It also allows for the parameterization of the Service Agent (the Receiver Interface, in Command Pattern terms). Therefore, implementations of the Command interface are composed of a Service Agent, a node object, and additional parameters as required. Service Agent objects include all of the logic required to complete the various operations (Restore, Clone, Initialize, and Measure) on a particular type of node, as dictated by the Service Agent interface. Service Agent's initiate an operation using Intel AMT which commences the network boot process. Once the node has booted into the remediation environment it establishes network communications with its Service Agent in order to commence an operation. The Service Agent then supports the node operation, providing the node with access to image data and verifying quotes received from the node. The Service Agent also communicates with the Conductor during operation by sending status events as well as a completion event which contains the final verdict of the integrity verification phase. The conductor then reports of the final status of its list of commands to the user or to an existing situational awareness framework in use on the network.

3.2.3 Node Software

The Remediation Environment is the environment in which all recovery related activities are conducted on client nodes. It is an operating system which runs entirely within a node's memory. In addition to requiring Operating System services in order to conduct remediation activities it also requires a credible chain of trust so that any measurement or reporting conducted in the environment can be trusted by a third party. This chain of trust is achieved using a CPU-based late-launch technique, such as Intel TXT, to create a DRTM and extend a chain of trust from the DRTM into the remediation environment. In order to provide a high level description of the Remediation Environment we will first describe the network boot chain required to establish it. The network boot chain is made up of a Network-Stage Bootloader and a Trust-Stage Bootloader which together bootstrap the system from the NIC firwmare to the Remediation Environment. Following this the required functionality of the environment will be described in greater detail. Figure 3.3 provides an illustration of the remediation environment and its associated boot chain.

As described, Intel AMT is used to configure the BIOS to conduct a PXE boot following the next restart. AMT is also used to initiate the next restart.

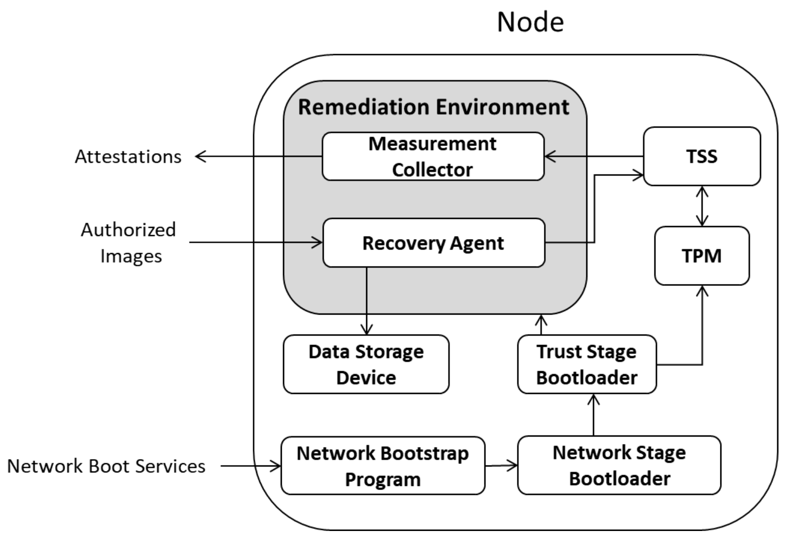

Figure 3.3: Design of Remediation Environment with supporting software

As part of the PXE boot process the NIC firmware retrieves and executes a Network Bootstrap Program. As mentioned in 2.1.2, the Network Bootstrap Program is small and only responsible for downloading further modules. In our case it is part of the Network-stage bootloader which is in turn responsible for downloading the Trust-stage Bootloader, the Remediation Environment, and additional files required to support the late-launch. The Network-stage Bootloader bootstraps the system from the PXE boot process executed by the NIC firmware to the Trusted-stage Bootloader that initiates the late-launch operation. The late-launch operation creates a DRTM which measures the Trust-stage Bootloader itself, the Remediation Environment, and CPU specific code modules used to support the late-launch. The hash values associated with these measurements are stored in special TPM PCRs which can only be extended by microcode associated with the late-launch.

The above process accomplishes two goals. First it utilizes the PXE process to boot a node into an environment provided by the Recovery Server. Secondly, it creates a chain of trust from a DRTM into the Remediation Environment runtime. Provided the DRTM is trusted (reasons why it might not be are covered in 6.1.2) then the remediation environment itself can be trusted.

The Remediation Environment provides two main functions. One is as a Recovery Agent which is responsible for carrying out operations on the node's disk in order to clone and restore the node. The other is as a Measurement Collector, which is responsible for extending PCRs with measurements from additional elements, such as disk images accessed by the Recovery Agent, as well as reporting system state, as represented by PCR values, to the Recovery Server.

As mentioned, the Remediation Environment itself is an Operating System running from memory. In order to act as a Measurement Collector the Remediation Environment must have certain configuration details for the TPM of the node it is running on. Additionally, the environment must have the required libraries to interact with the TPM. On Linux these libraries are provided by `TrouSers`, an open source implementation of the TCG Software Stack (TSS)[45].

3.3 System Operation

There are four principal node operations. These represent the four actions which can be conducted on nodes through the trusted recovery architecture. Node operations can be scripted to occur once the Remediation Environment boots, this allows the activity to be automated following the network boot process. This, along with the network boot process supported by the Recovery Server, fulfills Requirement 1.2, that the Framework be fully automated. The four classes of system operation will now be described in more detail in order to illustrate the use of the recovery framework.

3.3.1 Initialize & Measure

Initialize and Measure operations support the framework by providing the means to add new hardware and establish golden values for the BIOS and trust-stage bootloader modules. An Initialize operation handles all tasks required when adding new hardware to the network. Primarily this involves preparing the TPM to support the remainder of the framework's operations. Initialization activities occur within a generic Initialization Environment as opposed to within a Remediation Environment. Once The TPM has been initialized into the framework, all further operations occur through a node specific Remediation Environment which is configured for the node's particular TPM. Measure operations involve determining hash values for authorized BIOS configurations, trust-stage bootloaders and the Remediation Environments. These hash values are then stored on the Recovery Server as golden measurements.

Prior to the Initialize operation the TPM should be enabled in the BIOS and be in its reset and un-owned state. Once booted into an Initialization Environment, ownership is taken of the TPM. This involves setting an Owner and Storage password [29]. Next, a UUID, which will be used to identify the AIK in the TPM, is generated. Following this an AIK is created and loaded into the TPM NVRAM with the previously generated UUID. The products of these operations are the UUID, the public side of the AIK, and data used by the TPM libraries to facilitate TPM operations. These products are then sent to the Recovery Server. The TPM public key is stored in the node repository and used to verify the signature attached to quotes signed by that particular TPM. The UUID and TPM library data are incorporated in the build of a Remediation Environment specific to that node, so that the further TPM operations can be conducted on the node.

Measurement operations are concerned with measuring the trusted system state of the Remediation Environment so that this state can be verified when conducting future operations. Principally, this state is determined by the software modules in the chain of trust from the DRTM to the Remediation Environment runtime. These modules include the Trust-stage Bootloader which enacts the late-launch procedure, Authenticated Code Modules associated with the late-launch procedure (the SINIT ACM), and the Remediation Environment operating system files. As part of the late-launch procedure, as outlined by Intel and implemented in `tboot`, hashes of these modules are calculated with the resultant values being extended into specially designated PCRs. This means that in order to measure the late-launch trusted boot chain the TPM can be called on to generate a quote over these PCRs. This quote data is then sent to the Recovery Server, which verifies the signature using its knowledge of the TPM's public AIK, and then stores the value in the Measurement Repository.

The process to measure the BIOS unfolds similarly. In this case, the BIOS and its configuration are stored in PCRs zero through seven [29]. A quote is generated over these values, the signature is verified by the Recovery Server and they are stored in the Measurement Repository. The distinction between the measurement of the BIOS and the Remediation Environment is a key aspect of the framework proposed by this research. As mentioned in 2.3.1, a TPM quote involves a composite hash over a series of PCR values. As a result, it is impossible to attribute a change in the TPM composite hash to a specific PCR within it. By increasing the number of TPM quote operations and decreasing the number of PCRs in each quote, a greater granularity into system state is gained. However, as will be discussed in 5.2, TPM quote operations can be computationally expensive. The Framework considers the

BIOS and Trusted Boot Chain separately.

3.3.2 Clone

The Clone operation generates authorized disk images from a node in an operational and trustworthy state. Images can be cloned either from the entire disk or, more efficiently, at the partition level, by including only sectors the filesystem is using. Once cloned, the image(s) are sent to the Framework for storage. Images can be stored either on the Recovery Server or on a file server elsewhere on the network. In parallel with the storage of the images, the SHA-1 hash is calculated for each image. These values are used as golden measurements with which to verify the images as they are used to restore a node.

This operation satisfies Requirements 1.1 from 3.1, that the restoration maintain system specific configuration and software. This is because an image from an operational system preserves the software and all aspects of configuration.

3.3.3 Restore

The final system operation, Restore, performs the actual remediation of the node. The process carried out by the Recovery Server is shown in the sequence diagram in Figure 3.4 which depicts the Restore operation. While the sequence conducted by the ServiceAgent in the diagram is specific to the Restore operation, the other actions on the diagram are universal between operations and therefore their explanation in this subsection is applicable to all operations,

The client object in Figure 3.4 is responsible for interacting with the UI. When an operation is desired on a particular node a Node object is instantiated with data from the Database Layer (not shown in figure). A ServiceAgent object applicable to the node in question is also created. At this point an Add<operation>Command(...) function is called on the Conductor. The Conductor will create the appropriate Command and add it to its list of Commands to execute. When the Client calls Run() on the Conductor it will start Network Boot Services and call Execute() on each Command. Each Command will then execute asynchronously. The Command in turn will call the desired function on the ServiceAgent, in the case of Figure 3.4 the Restore() function is called. At this point the remediation of the node will begin.

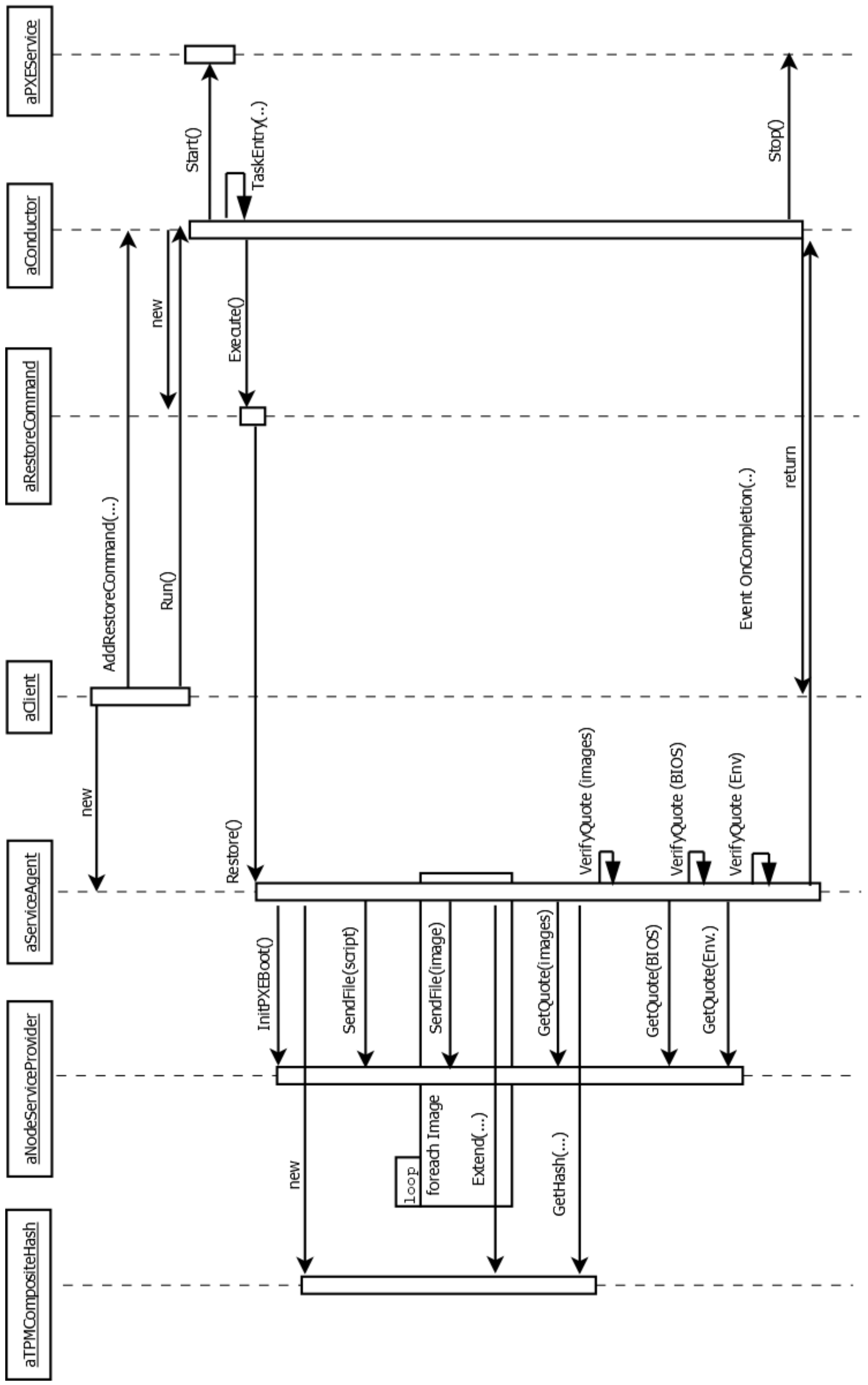

Figure 3.4: Sequence diagram for the restore operation conducted by the recovery server

As is the case with all node operations, the ServiceAgent object conducts the remediation. Each instance of a ServiceAgent uses high level node functions provided by a NodeServiceProvider class to interact with a specific node in the network. These high level functions include SendFile(), GetFile(), GetQuote(), InitPXEBoot() among other actions. A ServiceAgent will initiate the process by calling InitPXEBoot() which will configure the node to conduct a PXE boot on its next start up and execute a restart on the node using Intel AMT. This will cause the node to boot using the Network Boot Services provided by the Recovery Server and load the Remediation Environment. The Recovery Server will then send the applicable script to the node, in this case a script which conducts a system remediation. As the node receives the script it will, in parallel, extend its measurement into a PCR and write it to the file system. Once the file has been fully received it will be executed. The script will then conduct the remediation on the node.

The remediation happens in two phases. First, a node is restored to a trusted image, secondly, a series of integrity metrics from the node are reported on so that the Recovery Server can verify the remediation. The first phase is relatively straightforward. The Remediation Environment retrieves the applicable image(s) from the network. As the images are streamed over the network and written to the disk they are, in parallel, hashed and extended into a TPM PCR. These can then be verified at a later time. The order the images are streamed in is specific to the partition scheme used by the particular node.

Once a node has been restored, its integrity must be verified by the Recovery Server. This verification is vital because, if successful, it allows the operator to trust critical equipment to behave as expected. The integrity of a node following remediation is a function of the integrity of the Remediation Environment and its trusted boot chain, the integrity of the disk images restored to, and the integrity of the underlying BIOS firmware which persists following the remediation. As the process of assessing the integrity of these elements on a node is crucial to the Trusted Recovery Framework it is covered in some detail.

Verification Phase

The Recovery Server conducts the Verification Phase. That is, using the golden measurements stored in the Measurement and Image Repositories, the Server is responsible for verifying the integrity of a node following remediation and reporting the status of the verification to the network operator.

While the TPM quote process was briefly outlined in 2.3.1, it will be described again here in terms of the proposed Framework in order to make the subsequent sections more clear. A TPM quote is composed of a composite hash derived from a set of PCR values, some additional data specific to the quote, and a signature generated using the AIK. To prevent replay attacks, the quote also includes a nonce provided by the remote verifier, in this case the Recovery Server. Therefore, when conducting remote attestation, the Recovery Server generates and sends a psuedo-random nonce to the node to be verified. The node sends a message to the TPM with the nonce, requesting it quote a designated set of PCRs. The quote is generated on the TPM itself and signed using the private AIK, which is stored on the TPM NVRAM and is not accessible outside of the TPM. The Recovery Server then uses the public AIK to verify the quote data and the identity of the node that signed it, as the AIK is unique to the TPM. It also verifies that the nonce contained in the quote matches that which it sent initially and finally, verifies that the composite hash of PCR values matches a golden value held on the Recovery Server. As a consequence of the above process, provided both the signature and nonce associated with a quote are verified, the Recovery Server can trust the quote data is an accurate representation of the current node state. This is important because as stated in our threat model in 2.2: both the network and the node prior to the remediation are not trustworthy. The remote attestation process allows trust to be reestablished in the node.

The Framework conducts the verification in three stages, each identifying certain aspects of the remediation process. Stage one verifies the disk images and any other elements received over the network, stage two verifies the BIOS firmware and its configuration, and stage three verifies the Remediation Environment and the Trusted Boot Chain. As elaborated on in 3.3.1, the TPM quote process and its associated verification are expensive operations. As a result the above three stages were chosen as a compromise between added granularity in detecting mis-configuration or tampering with the remediation process and increasing the resources required to recover the system.

Any source images used in the restoration as well as any additional inputs which may also be transferred are also susceptible to manipulation by the attacker described in our threat model. This means that they must be verified by the Recovery Server in order to trust the node following restoration. This verification differs from the verification of the Remediation Environment as the inputs to be measured are very likely to change independently during the operation of the network, for example, as a result of software updates. To accommodate these changes the Recovery Server maintains records of golden measurements for all potential inputs, be they disk image hashes or hashes of other data sent during the remediation. During the verification phase, the Recovery Server calculates its own composite hash based on its record of golden measurements, in the same fashion as the TPM does when conducting the quote (as seen with the TPMCompositeHash object in Figure 3.4). This allows the expected value of the TPM quote to be created dynamically based on what data was sent to the node during the operation.

The second aspect to be verified is the BIOS. This is important because the BIOS configuration may have been changed in such a way as to make the node more vulnerable. The BIOS may also be infected with a bootkit (as detailed in 2.1.3) which would not be addressed by the remediation process incorporated in this framework. As a result of the late-launch process the Remediation Environment is both separate from and protected from the system state previous to the late-launch. This makes it an ideal environment for measuring the integrity of the BIOS. While nothing is done to remedy the BIOS itself in this framework, it is capable of detecting changes. The integrity of the BIOS can then be signalled to the network operator, allowing them to make an informed security decision.

The final aspect verified is the Trusted Boot Chain & the Remediation Environment. This verification is vital as the Remediation Environment itself must be trusted in order to trust the outcome of the remediation. Additionally, the chain of trust from the DRTM, as captured by the Trusted Boot Chain measurements, must be verified too. Although the source of the modules that make up the Trusted Boot Chain and the Remediation Environment is the Recovery Server itself, they are transferred over the network, and as outlined in 2.2, the attacker has control over the network. The Trusted Boot Chain and the Remediation Environment are verified by comparison to the golden measurement stored on the Configuration Management Server. If this verification is successful we now trust the Remediation Environment to reliably report on the subsequent verifications, that of the elements used in the restoration and of the BIOS firmware.

Each verification is carried out in sequence following the restoration of the node, as depicted in Figure 3.5. If any verification fails the sequence ends and the failure is indicated to the operator. Since the restoration activities happen prior to the verification sequence, this failure does not necessarily mean that the node is not operational. This allows the operator the option of accepting the risk of returning the node to operation in order to meet service demands despite a failed verification process.

Following the verification phase the node is rebooted. On the Configuration Management Server a completion event is sent to the Conductor object, which is then, in turn, responsible for reporting the outcome of the operations

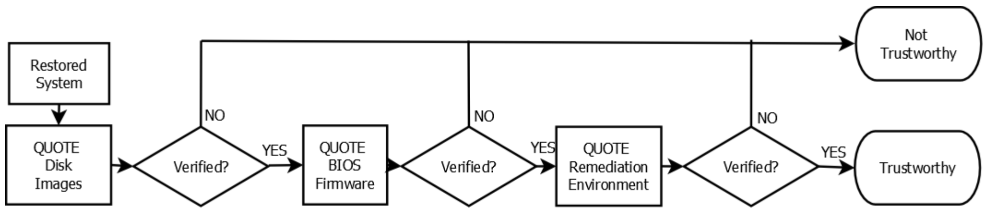

Figure 3.5: The verification phase control flow

to the Client. On the node, the BIOS reverts to its previously configured boot device order, which means the node will boot from its newly restored hard disk. The node is now operational and can be trusted to the extent to which the verification phase was successful. The result of the verification phase indicates to the network operator the degree to which the remediation process has been successful at addressing persistent malware or an adversary who attempts to interfere with the recovery process, the two principal threats identified in 2.1.2 and 2.1.3. When it comes to rootkits & bootkits there are three broad cases:

1. The rootkit is remediated but not detected
2. The bootkit is detected but not remediated
3. The bootkit is not detected or remediated

These cases are mapped to various types of root and boot kits in table 3.2. The proposed framework does not achieve the ideal case of both detecting and remediating a rootkit or bootkit infection. This stems from the unique priorities of SCADA systems. The timely restoration of supervisory control over an Industrial Control System trumps other concerns, such as forensic investigation. In order to realize this, the framework prioritizes restoration activities over integrity measurement, therefore the best case outcome is for the framework to have removed malware without having detected it. The next best scenario is for the framework to detect a bootkit but not remediate it. This occurs when the bootkit resides in the BIOS firmware. In this case, the BIOS integrity is measured as part of the verification phase, and the result is reported to the network operator. At this stage the network operator is able to investigate further and respond accordingly. The worst case scenario occurs if an adversary manages to install a bootkit in system firmware which is not measured as part of the verification phase. Potential locations include the NIC firmware, hard drive firmware, or even, potentially, the Intel ME firmware. These techniques indicate a highly advanced, likely nation-state level, adversary. In this case the bootkit is neither detected nor removed.

Table 3.2: The extent to which potential rootkit or bootkits are addressed

Rootkit/Bootkit Location	Detected	Remediated
OS (User-mode)	No	Yes
OS (Kernel-mode)	No	Yes
Boot Sector (eg. MBR)	No	Yes
BIOS/UEFI	Yes	No
System Firmware (other than BIOS)	No	No

The framework is also resilient to manipulation by an adversary. As explained in 2.2 the adversary is given full control over the network. This allows them to interfere with communications between the node and server by creating, injecting, or modifying packets in the communication stream. Using this ability the adversary could corrupt modules and images in transit in order to conduct a denial of service attack by preventing the remediation from occurring. The framework is incapable of countering a denial of service attack of this nature. However, if the adversary attempts to manipulate the process in order to maintain persistence on a node following remediation the framework will necessarily detect them. Attempts to modify the Trusted Boot Chain or Remediation Environment will be detected during the first stage of the verification phase. The image files in transit are equally resistant to this type of manipulation. Since the integrity hash data for the images are calculated on the node, within the Remediation Environment, as long as the first stage of the verification phase was successful the image integrity measurement can be relied upon.

Table 3.3 summarizes the possible outcomes of the verification process and their implications. The first column lists some attack scenarios. The next three columns detail whether each stage of the verification process has resulted in a pass (P) or fail (F). These stages verify, from one to three in order, the late-launch trusted boot chain and Remediation Environment, all further inputs to the restoration process such as images, and the BIOS firmware. The final column lists the implications the operator can infer from the different results. The exact effects of an adversary's manipulation of network communications are difficult to enumerate as potential adversary actions vary widely. In general, a denial of service will leave the node not operational. Importantly, the framework prohibits any manipulation of the process resulting in all three stages of the verification phase being successful. More subtle techniques may not impede the restoration itself, but will cause the verification to fail, leaving the node untrustworthy.

Table 3.3: Attack scenarios and their implications on node availability and integrity

Scenario	1	2	3	Implications
1. User-mode Rootkit in OS	P	P	P	Node is operational and trustworthy
2. Kernel-mode Rootkit in OS	P	P	P	Node is operational and trustworthy
3. Bootkit in Boot Sector	P	P	P	Node is operational and trustworthy
4. Bootkit in BIOS	P	P	F	Node is operational but not trustworthy BIOS has been changed and potentially contains a bootkit
5. Bootkit in other Firmware	P	P	P	Node is operational but not trustworthy. Note: This framework is incapable of detecting bootkits in system firmware other than the BIOS. This scenarios is indistinguishable from scenarios one through three.
6. Adversary manipulates network communications	F	F	P	Node is not operational or trustworthy.

3.4 Design Summary

Section 3.1 provides a representative model of the SCADA networks we seek to recover. It also outlines the threat model and the ramifications on the design of a trusted recovery framework. As a result of those two models a list of requirements for the trusted recovery of SCADA networks was enumerated. This chapter has presented a design for a Trusted Recovery Framework for SCADA systems. Table 3.4 presents a summary of the requirements stated in 3.1 and lists which aspects of the design presented in 3.2 satisfy them. As can be seen, this framework satisfies all of the stated requirements. The next chapter will describe the implementation of a proof-of-concept framework for a target system used at sea in the Royal Canadian Navy.

Table 3.4: Fulfillment of the high level requirements by the proposed trusted recovery framework design

No.	Requirement	Design Aspects
1.1	Restoration must maintain system specific configuration and software	Restoring nodes from images which have been cloned from operational systems preserves system functionality (3.3.2).
1.2	Framework must be fully automated	Automation is achieved through the use of AMT initiated network booting (3.2.2) and shell scripts tied to the Remediation Environment's init system (3.2.3).
1.3	Framework must address persistent malware	Persistent malware is addressed by restoring the entire disk (including the MBR)(3.3.3) and the conduct of integrity measurements of the BIOS (3.3.3). Note: this framework does not address bootkits in firmware other than BIOS
1.4	Framework should fail to an operation state when ever possible	Although failure during a disk write procedure will leave a node in an unstable state, failure at any time during the integrity verification phase will not impede the restoration of the node (3.3.3).
2.1	Framework must not trust network	As a result of CPU-based late-launch and the integrity verification phase an attacker with control of the network cannot forge the attestation of trust provided by the framework (3.2.3, 3.3.3)
2.2	Framework must not trust node software	As a result of network bootstrap process and CPU-based late-launch the system state previous to remediation has no bearing on the outcome of the remediation. (3.2.2, 3.2.3)

4 Implementation

In the first chapter, we described the importance of being able to recover SCADA networks following a computer network intrusion. In the second chapter we expanded on the problem of recovering computer networks in a trusted manner when in the presence of an adversary and their malware. We also introduced the concept of Trusted Computing and provided background on hardware security capabilities which support it. The third chapter proposed a design for a Trusted Recovery Framework which is capable of addressing the problem of recovering a SCADA network following a compromise. This chapter will describe a proof-of-concept implementation of the framework for the Integrated Platform Management System (IPMS). IPMS is the SCADA network used by the Royal Canadian Navy to supervise the machinery plant of several classes of warship. While both the maritime and military environments are unique, SCADA networks are commonly deployed in unique conditions, and the general models described in Section 2.1 and provided by NIST are general enough to apply in these environments [4]. Therefore, for the purposes of validating our proposed design of a Trusted Recovery Framework for SCADA systems, we consider IPMS a representative target system. This proof-of-concept implementation will be used to validate the proposed design in the following chapter.

The first section of this chapter will provide a brief description of the IPMS network as deployed on the Canadian Patrol Frigate(CPF). Following this it will describe the laboratory configuration used to develop and validate the framework. The final two sections will contain details of the implementation for the node-based Remediation Environment and the Recovery Server.

4.1 The Integrated Platform Management System

IPMS is a typical SCADA network in that remote sensors, actuators, and machinery controllers are interfaced to it through Remote Terminal Units

(RTUs). The RTUs are connected to 2 redundant TCP/IP based networks. These networks then connect to each of the SCADA nodes. Nodes come in several types, including: consoles, large screen displays (LSDs), and local operating panels (LOPs). IPMS also provides numerous ethernet jacks throughout the ship for portable operating units (POUs), essentially laptop computers, to be connected. The consoles, LSDs, LOPs, and POUs run the IPMS software which provides the graphical interface for control and monitoring of the machinery plant. Technicians use the IPMS interface to start or stop machinery such as pumps, generators, or the gas turbine engines. Operators use the interface to make changes to the ordered speed or propeller pitch while maneuvering the ship. The interface includes the ability to monitor the status of the propulsion, electrical, auxiliary, and damage control systems. In addition, the system also provides a damage control incident board which is used to coordinate the damage control efforts of numerous outstations during shipboard emergencies. For these reasons IPMS is vital to operations onboard a CPF; ship's staff must be able to trust IPMS in order to operate effectively at sea.

At the network level, all of the IPMS nodes sit within a dedicated subnet on several redundant networks. At a hardware level the nodes are ruggedized versions of commodity computer hardware which include modern Intel chipsets, UEFI firmware, and NIC functionality. Most, but not all, nodes support Intel AMT and its remote administration capability. At the software level IPMS runs within the Windows Operating System, including versions XP, 7 , and 10. For the purpose of development and validation an experimental environment which consisted of representative and actual IPMS hardware was established.

4.2 Development & Validation Environment

The proof-of-concept implementation was developed using actual and facsimile hardware. The IPMS vendor and project office made several laptops (POUs) available for testing and development. In order to spare this hardware the wear of prolonged power cycles and disk writes during testing, a consumer grade laptop with comparable hardware and software was used during stages of both development and validation. The IPMS laptop used was a DELL Latitude E6430 ATG, a semi-rugged version of the Latitude line of Dell laptops running Windows XP. The non-IPMS target was a Lenovo Thinkpad T420 running Windows 7. While the Thinkpad is not used in IPMS, it has equivalent hardware security capabilities, in particular a TPM, a chipset with supports late-launch, and Intel AMT. The Recovery Server was developed and tested on a System 76 Lemur laptop. System images were stored on an external solid state drive which could accommodate large images sizes and also be secured in accordance with IPMS handling policies.

The development targets and the server were either connected on a point-to-point basis for testing against one target or through an Ethernet switch to test simultaneous node operation. This set up simulates either a static infrastructure where the Recovery Services are provided by an existing node, such as a Configuration Management Server, or a scenario in which the Recovery Server is connected to the network, or partitions of the network, only when supporting a recovery effort.

4.3 Recovery Server

The Recovery Server was developed in a Linux environment using C# and the Mono project, an open source implementation of the .NET Framework. While minimal consideration was given to portability in the course of this research, the cross-platform nature of the Common Language Runtime means it is likely straightforward to port aspects of the server to Windows. To streamline debugging the server was given a Command Line Interface (CLI) but the framework is intended to be incorporated into existing node management software or deployed with a more user friendly interface. Using the CLI of the proof-of-concept implementation, a user is able to list nodes and their details as well as call for nodes to be cloned or restored.

4.3.1 Network Boot Services

Section 3.2.2 provides an overview of the services required to support the automated network booting of nodes. These services include Intel AMT functionality and PXE (DHCP and TFTP).

The Recovery Server's application layer accesses Intel AMT features through a wrapper class, which was designed to provide high level access to various node functions, including Power & Boot Operation. This wrapper class uses Intel's AMT Software Developer's Kit (SDK) to provide Intel AMT functionality. The Intel AMT SDK provides a High-level Application Programming Interface (API) for the Intel AMT [46]. This API provides direct support for all AMT features in C# as well as a COM interface for a handful of features in other languages [46]. The Recovery Server only required support for basic Boot & Power operations which are readily implemented using the SDK.

The Recovery Server was programmed to start and stop a third party PXE server as required during system operations. In this case, the PXE server used was `dnsmasq` [47]. The framework utilizes only basic DHCP and TFTP functionality and the required services could potentially be implemented into the framework with additional effort, removing the requirement for 3rd party software. `Dnsmasq` provides a lightweight and easily configurable PXE server. The DHCP server was configured to provide static IP addresses to the nodes, based on MAC address. These IP addresses correspond to the IP address normally assigned to the node in IPMS' operation. The TFTP server was configured to serve the `PXELINUX` bootstrap program to the node during the PXE boot process. `PXELINUX` will be discussed in Section 4.4.1.

4.3.2 Data Layer

The Data layer of the Recovery Server includes interfaces for the required data repositories. Each interface specifies the operations required for the particular data in each repository. In this implementation the interfaces were implemented using a `SQLite` database [48]. `SQLite` is a lightweight database engine designed to run within the application process. The `SQLite` library is distributed as a single file which is linked in to the application program during compilation. Given its lightweight nature and the ease of configuration it was ideal for this project. The data layer also makes use of `Dapper` [49], an Object-Relational Mapping library. `Dapper` allowed for the data stored in the repositories to mapped to data objects within the application layer. This greater simplified the act of accessing and storing data in the framework.

4.3.3 Application Layer

As seen in Figure 3.2 the application layer is straightforward to implement in an object-oriented language. In addition to the implementation of the classes specified in Figure 3.2 further classes were included to store Node and Restore Point data. Node objects provided a straightforward way to encapsulate and pass the data for a particular node through the software. Restore Point objects encapsulate the resulting image locations and metadata associated with a clone operation. Therefore, when a user requests a node be restored to a certain restore point, a Restore Point object can be passed to the Restore command along with a node object representing the node to be restored.

4.4 Node-based

This section details the implementation of the node-based portion of the Trusted Recovery Framework. This includes the network stage bootloader, the trust stage bootloader, and the remediation environment.

4.4.1 Network Stage Bootloader

The network stage bootloader is required to bootstrap from the NIC firmware into the trust stage bootloader. For this project `PXELINUX` [50], was chosen as the network stage bootloader. `PXELINUX` is provided under the GNU General Public License (GPL) Version 2 and provides the ability to boot Linux from a PXE server. It consists of an image, the Network Bootstrap Program, which in turn grabs an intermediate bootloader which is in turn used to boot the operating system. In our case it is used to chain load the trust stage bootloader. `PXELINUX` can be configured to boot nodes to different images based on their MAC or IP address. The framework uses this feature to boot each node into an operating system based on its type. This allows the framework to accommodate nodes with different hardware configurations. `PXELINUX` downloads the trust stage bootloader, supporting modules, and the remediation environment into memory before executing the trust stage bootloader.

4.4.2 Trust Stage Bootloader

The trust stage bootloader is responsible for conducting the late-launch procedure to execute the remediation environment. For this implementation Trusted Boot [51] (`tboot`) was used. `Tboot` is an open source bootloader which uses Intel TXT to launch either a hypervisor or a Linux kernel. It makes use of an SINIT ACM provided by Intel to conduct a late-launch in accordance with a user defined Launch Control Policy. Following the late-launch it calls `kexec` on the Remediation Environment's kernel, turning control over the Remediation Environment.

4.4.3 Remediation Environment

As mentioned, the Remediation Environment is an operating system which has been configured and scripted to support the remediation process. In this implementation, a custom kernel and initial RAM disk were built using Buildroot [52]. Buildroot is a cross-compilation tool for creating embedded Linux systems. It provides a ncurses menu through which the system can be configured to meet a variety of needs. It allowed the Remediation Environment to be built with a footprint of less than 20MB. This small footprint is valuable for several reasons. First of all it greatly decreases the amount of time required to both stream the images over the network and the time required to calculate the SHA-1 hash. An added benefit of the small size is that it can reliably run on memory constrained systems, as are likely to found on ICS networks. Buildroot also offers a build system which includes numerous packages and methods of adding new packages to the system. It also allows a custom root file system overlay to be specified which is then reflected in the resultant system file system. Finally, it includes support for many architectures in addition to x86_64 which means the recovery OS is extensible to additional hardware.

The kernel is Version 2.4 of the Linux kernel. The only customization to the kernel was the inclusion of kernel modules to support the TPM. The `TCG_TPM` module is required for basic TPM communications, additional modules, such as `TCG_NSC`, `TCG_ATMEL`, and `TCG_INFINEON`, are required to support the different vendor implementations of the TPM. This allows one kernel to be compatible with multiple types of hardware, provided they are of the same CPU architecture. The initial RAM disk, however, was built for a specific node. This is because the library required for TPM operations, `TrouSers`, must be configured for the node's specific TPM.

Figure 4.1 depicts the system and the programs which it is composed of. Bash was used to script the Initialize, Measure, Clone, and Restore operations. Initially `ash`, a lightweight linux shell included with Buildroot, was used. Unfortunately it did not support process substitution, which was required to pipe the output of one command to the input of two (or more) other commands. This operation was required to support parallelism.

File and data transfer was handled using `netcat`, a simple program for sending and receiving arbitrary data through TCP connections. TPM operations were supported by `TrouSers`, `tpm-tools`, `tpm-quote-tools`, and `pcr-extend`. As mentioned in the previous chapter, `TrouSers` is an open source implementation of the TCG Software Stack (TSS). It provides `tcsd` a user space daemon which provides access to TPM commands. `TrouSers` includes the `tpm-tools` package and is used by `tpm-quote-tools` and `pcr-extend`. `Tpm-tools` provides basic TPM administration functions that allow the framework to take take ownership of the TPM and configure it for use. The `tpm-quote-tools` package extends this functionality with programs to generate AIKs as well as to generate and verify TPM quotes. `Pcr-extend` is a small software tool which calculates the SHA-1 hash of predefined data and extends a user designated PCR with the result. In the course of implementation `pcr-extend` had to be modified to read and hash data from standard input as opposed to from a file. While `TrouSers` and `tpm-tools` were included as packages in Buildroot, custom make instructions had to be developed in order to compile `tpm-quote-tools` and the modified `pcr-extend` for the remediation environment. Most of the configuration changes were required because the remediation environment makes use of μlibc, a small C library for embedded systems, as opposed to the standard C library.

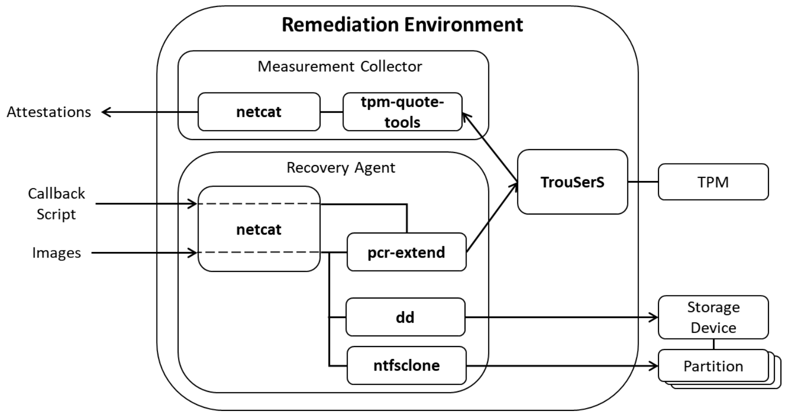

Figure 4.1: Implementation of the remediation environment

Now that the software which makes up the Remediation Environment has been described, a short description of the Restore operation from the point of view of the node will be provided. Once the Remediation Environment has finished booting the init system executes a series of scripts to start essential services, including the `tcsd` daemon. At this point the node has acquired an IP address from the DHCP service on the Recovery Server. As mentioned, this IP is statically assigned based on the hardware address. The final script executed as part of system initialization uses the node's IP address to determine a port number to call back to on the Recovery Server. The node initiates a TCP connection on the Recovery Server at the determined port number using `netcat` and begins to receive a script. Since the script is received following the node's callback to the server, we refer to it as the `callback` script. As the *callback* script is received its data is piped to both `pcr-extend` and to a file. `Pcr-extend` reads the script from standard input and generates a SHA1 hash which is extended into a TPM PCR. Once the transfer is completed the script is executed. Since a record of the script now sits in a PCR, it can be verified during the verification phase. In this example we are describing the Restore operation, so the `callback` script undertakes the Restore operation. It first receives a compressed image of the MBR from Recovery Server, decompresses it, and writes it to the storage device using `dd`. As with the previous network transfer, the data received from the Recovery Server is also piped to `pcr-extend` so it can be recorded in a PCR. Overwriting the MBR allows the partition table to be reestablished (if damaged). Once the partition table is reestablished, New Technology File System (NTFS) partitions can be restored using `ntfsclone`. `Ntfsclone` creates and writes partition images based on which blocks contain data allocated by the file system, as opposed to cloning the entire disk block by block. This allows both the clone and restore procedures to be conducted considerably faster. The choice of using `ntfsclone` is discussed at greater length in 6.1.2. The node iterates through this process for each partition on the disk. Once completed the node has been restored and the verification phase begins. The verification phase has been treated at length in the previous chapter. It suffices to say that `tpm-quote-tools` is used to generate TPM quotes using fresh nonces provided by the Recovery Server. As previously described, three quote operations are conducted, one for each stage of the verification process. The *callback* script and each image used during the restoration have all been measured and extended into PCRs on the node. Since the Recovery Server knows what script it sent to the node and the hashes of each image provided, it can dynamically determine the authorized quote value for the second stage of the verification phase. Following the verification phase the system is rebooted into its recovered state.

This chapter has described a proof-of-concept implementation of the framework design presented in the previous chapter for IPMS. It first described the target system, IPMS, and the environment in which the implementation was developed. It then enumerated the various software and libraries which were used to implement the framework and described how the implementation worked in practice. The next chapter will analyze the proof-of-concept implementation in terms of the design requirements derived in Chapter 3 and the deficiency outlined in Chapter 1. Through this analysis it will show that analysis of the implementation described above demonstrates that the proposed Trusted Recovery Framework satisfies the aim of this research.

5 Validation

Chapter 1 introduced the problem of recovering SCADA systems to a trustworthy state following a computer security incident and described how this research aims to address the problem. Chapter 2 elaborated on the threat posed to potential recovery frameworks by malware which interferes with the recovery process and bootkits installed at low levels on the system. The second chapter also described hardware security capabilities embedded in modern computing systems and how they can establish the trustworthiness of system. Chapter 3 presented the design for a framework which leverages those trusted computing capabilities in order to restore a SCADA system to a trustworthy state. Chapter 4 detailed a proof-of-concept implementation for a representative SCADA network, the Integrated Platform Managment System. This chapter will argue that the design for a Trusted Recovery Framework for SCADA systems presented in Chapter 3 satisfies the aim of this research stated in Chapter 1. First, the implementation described in Chapter 4 will be shown to meet the high-level requirements outlined in Table 3.1. This will show that the proof-of-concept framework is an accurate implementation of the proposed design. Following this the implementation will be analyzed in terms of the Statements of Deficiency and Aim to demonstrate how the design for a Trusted Recovery Framework is capable of leveraging inherent embedded hardware security capabilities in order to recover SCADA systems to a trustworthy state and consequently, that the aim of this research has been satisfied.

5.1 Analysis

The analysis begins by empirically demonstrating how the implementation satisfies each of the high level design requirements enumerated in tables 3.1 and 3.4. This both establishes that a functional implementation of the proposed design is possible and that the proof-of-concept system is a genuine representation of the design. Once this has been established the proof-of-concept is evaluated against the stated deficiency and the aim of this research.

5.1.1 Verification of the Proof-of-Concept Implementation

Tables 3.1 and 3.4 contain the high level requirements and how aspects of the design seek to address them. They are included again in Table 5.1 for reference. Testing has shown that the proof-of-concept implementation realizes each of these. While in many cases it is evident from the description of the technique that the requirement is satisfied, for completeness each was specifically tested and is covered in this section.

Table 5.1: Trusted Recovery Framework Requirements (Repeated from 3.1)

No.	Requirement
1.1	Restoration must maintain system specific configuration and software
1.2	Framework must be fully automated
1.3	Framework must address persistent malware
1.4	Framework should fail to an operation state
2.1	Framework must not trust network
2.2	Framework must not trust node software

Requirement 1.1, that system specific configuration and software must be preserved during recovery was trialed using both the IPMS POU and the facsimile target. In testing, the implementation's technique of copying the node's MBR and then creating NTFS images of subsequent partitions during the clone operation was able to preserve the system state. Once the partition table is reestablished and the NTFS partition images have been written to each partition the system state is identical to that of the system when cloned. This was verified during testing by observing that `ntfsclone` was behaving as expected and the newly restored system reflected that desired system state.

Requirement 1.2 stated that the framework must be fully automated. This is easily verified in testing. Once launched from the Recovery Server CLI, any operation requires no further operator input, with the exception of the Initialize operation which requires the node specific remediation environment to be built on completion. This action is taken outside of the described framework and involves using `Buildroot` to generate an initial RAM disk which includes node specific `tcsd` library files generated during the TPM ownership process. This step is required so that subsequent node operations can make use of the TPM.

Requirement 1.3 requires the framework to address malware persistence at the lowest level possible. The design addresses this differently depending on the level in question. For the same reasons that system specific configuration and software are preserved by the process, bootkits which reside on the storage disk, either in the boot sector or operating system, are necessarily overwritten as part of the restore procedure. This is seen during testing when corruptions added to the MBR or changes to the operating system during testing are negated by the Restore operation. During testing it was confirmed that the framework is also capable of detecting configuration changes to the BIOS. This was verified by observing the BIOS verification stage fail following changes to various BIOS settings. This simulates detection of a BIOS bootkit as changes to the BIOS, be they configuration changes or the installation of a bootkit, will both end with a resultant change to a PCR value which will cause the verification to fail. This last supposition only holds if the BIOS in question implements total code coverage in its trust chain. If aspects of the BIOS are not measured as part of the trusted boot process, then modifications can potentially evade the verification process. Addressing issues with individual BIOS implementations is outside the scope of this research.

Requirement 1.4, that the framework should fail to an operational state was also verified during testing. During testing it was observed that as long as disk write operations complete the node will be operational, to the extent that an adversary has not interfered with its availability.

Requirements 2.1 and 2.2 are based on what facets of the network the framework can not rely on to bootstrap trust. In order to verify that the implementation can operate within these constraints we will examine the techniques the framework uses to generate integrity. While the previous requirements (1.1-1.4) were verified empirically, we present a rational argument for the satisfaction of requirements 2.1 and 2.2. This is because empirically verifying these properties is a prohibitively complex and onerous task. To verify that an adversary with control of the network or a node is unable to spoof the results of the verification phase requires an implementation of adversarial techniques which attempt to spoof authorized values on a compromised node. Such techniques will need to be implemented for an adversary on the network as well as on the node. It must then be argued that the techniques provide a sufficient representation of potential attacks to the system.

As opposed to this approach, we accept the results of Parneo, et al and Maene, et al when evaluating trusted computing technologies and architectures [27][43]. We then argue that the proof-of-concept system implements each of the trusted computing technologies in such a way as to provide a credible guarantee of integrity.

The reader is reminded of the formal specification of an integrity chain in equation 2.1. That is, that the integrity of a level in an integrity chain is a function of the integrity in the preceding level and the result of a verification function over that level. The integrity of the root level is assumed. The framework will be analyzed using this equation in order to formally qualify the integrity of the recovery process. Since the root level, level 0, must be assumed to be trustworthy it is the starting point of this analysis. In Trusted Computing, level 0 is the Root of Trust for Measurement. The proof-of-concept implementation uses Intel TXT to create a DRTM. This DRTM is created when the microcode associated with the late-launch CPU instruction measures the SINIT ACM. Since the SINIT ACM is digitally signed by the vendor, it is also verified by the CPU. In this way the DRTM is created. The SINIT ACM, in addition to implementing certain protections, measures the `tboot` boot loader before passing execution back to it. `Tboot` measures the Remediation Environment kernel and initial RAM disk, completes the protection measures, and passes execution to the kernel. There is one more condition that must be met in order to ensure the integrity of the remediation environment. The requirements stipulate that the framework must not rely on the integrity of either the network or the node prior to the operation. The integrity of the node is compromised because of our threat model and the attacker's access described in our threat model. The integrity of PXE boot process, specifically the network-stage bootloader is potentially compromised by an attacker on the network's ability to man-in-the-middle its transfer. For the purposes of this research, the protections implemented by Intel TXT are sufficient to isolate the integrity chain from the DRTM up to and including the remediation environment from the system state prior to late-launch. If they were not, equation 2.1 does not hold and the integrity of the recovery can not be verified. The extent to which Intel TXT can be relied on will be discussed in 6.1.2. The first stage of the verification phase attests to the integrity of the remediation environment and its integrity chain through the late-launch process. Once this integrity has been established the remediation environment can be trusted to establish the integrity of subsequent elements. The process by which subsequent aspects of the restoration are verified has been previously described in detail in this paper. The proof-of-concept system implements the verification phase control flow depicted in figure 3.5 with the use of `goto` statements on the result of a failed verification stage. Only once all three stages are verified is the trusted status variable set to true. Additionally, events are triggered for each stage on both successful and failed verifications. In this way an accurate picture of the state of a node during the restoration process is captured by the Recovery Server.

5.1.2 Validation of the Design

Now that the proof-of-concept implementation has been verified against the design requirements generated in section 3.1 we seek to validate the design against the stated aim of this research. The stated aim of this research was to design a framework which leverages embedded security features in modern computer systems in order to conduct a trusted recovery of a SCADA system following a computer security compromise. This aim was expanded upon with a description of relevant characteristics of SCADA networks and concerns specific to the recovery of SCADA systems. From these descriptions and an associated threat model, a set of formal high-level requirements were generated. As a result, the set of high level requirements flow directly from the stated aim of the research and must be satisfied in order for the aim itself to be achieved. In the course of this research a Trusted Recovery Framework which utilized security capabilities embedded in modern computer systems was designed in order to meet those requirements. This design was realized in a proof-of-concept implementation for a representative SCADA system, IPMS. In the previous subsection the proof-of-concept implementation was verified against the design requirements using both empirical testing and rational arguments. This implies that the proof-of-concept is an accurate implementation of the proposed design. Consequently, the implementation is validation that the design for the Trusted Recovery Framework satisfies the aim of this research.

5.2 Performance Characteristics

Following the validation activities, the performance of the architecture in the laboratory configuration was examined. Given the emphasis on availability in SCADA networks, the ammount of time required to conduct a restore operation was measured. Broadly speaking, the length of this operation is composed of the time required to boot the remediation environment, restore the authorized image, and conduct the verification phase. Of these steps, the restoration phase takes the longest. The length of time required to restore a disk image is a function of the size of the disk image and the speed that the process can attain. The size of the image is greatly influenced by the choice of using `ntfsclone` to create an image of the allocated portion of the New Technology File System (NTFS) partition which does not include unused disk space. The dramatic change in size can be seen in tables 5.2 and 5.3, which depict the actual disk space utilized on the two target nodes. Table 5.2 shows

Table 5.2: Dell E6430 ATG Disk Utilization

Partition 1		
Size	Utilization	Notes
298 GB	8.4 GB	Windows XP + IPMS Software

Table 5.3: Thinkpad T420 Disk Utilization

Partition 1		Partition 2		
Size	Utilization	Size	Utilization	Notes
105 MB	26 MB	298 GB	15.6 GB	Windows 7 64 + no additional software

that given modern system disk sizes, the operating system and IPMS only occupy a small fraction.

While potential drawbacks of using `ntfsclone` are discussed in 6.1.2, it is extremely beneficial when considering the time required to restore or clone a node. The difference in speed this allows compared to the use of `dd` was bench marked using the Thinkpad T420 with a Windows 7 installation which occupied roughly 15 GB of the 300GB hardrive. The `dd` program took 4:05 hours to clone and compress the disk and 1:15 hours to uncompress and restore the disk. After compression the image created by `dd` was 54 GB. Conversely, `ntfsclone` took 6:32 minutes to clone the Windows partition with a resultant image size of 14 GB and 6:11 min to restore the image. While compression was likely the limiting factor in the use of `dd`, the benefits of `ntfsclone` are clear.

This data was captured with the node connected directly to the Recovery Server and the image being stored directly on the Recovery Server's solid state drive. The trusted recovery framework performance was measured in the development configuration, with nodes connected to the Recovery Server through an Ethernet switch and the images stored on an external solid state drive connected to the Recovery Server. In this configuration, it is expected that the speed of read and write operations on the external solid state drive became the limiting factor, although this was not confirmed as it was seen as ancillary to the aim of the research. In a production environment this framework would likely not have this constraint. Table 5.4 contains performance data for each of the nodes. In order to eliminate interference between simultaneous operations this data was measured during the restore operation of each

Table 5.4: Time Required for the Restore Operation

Node Type	Reboot	PXE	tboot	OS Boot	Restore	Verification
Dell E6430 ATG	8s	14s	8s	8s	12m 9s	14s
Thinkpad T420	8s	18s	6s	5s	22m21s	18s

node individually.

The reboot stage consists of the time span from when the operation is commenced on the Recovery Server and the system power is cycled on the node. The PXE stage is the span from the power cycle to the execution of `tboot`. The remainder of the stages are self explanatory. Since the reboot phase consists mainly of the Recovery server using AMT to remotely configure the BIOS and initiate the power cycle, it is expected that it would take the same amount of time regardless of node type. The differences in NIC and BIOS firmware implementations account for the differences in time required to boot into the operating system. The differences in time required to restore each node correspond to the differences in utilized disk space on each node. The verification phase was limited by the speed of TPM operations. The differing TPM implementations and drivers explains the difference in time required for equivalent operations on each node. Generating the PCR composite hash data took approximately half of the duration with the signing of the data taking the other half.

This chapter has analyzed the proof-of-concept implementation by verifying its operation against the formulated high level design requirements. It then went on to argue that the proof-of-concept implementation demonstrates that the design for a Trusted Recovery Framework presented in Chapter 3 satisfies the stated aim of this thesis. This chapter followed up this argument with a discussion of the performance characteristics of the framework in a laboratory setting. The next chapter will provide further discussion into limitations of the design and potential future work in this area.

6 Discussion & Conclusion

Previous chapters have laid out the SCADA recovery problem, trusted computing technologies which can be utilized to address the problem, and a proposed design for a trusted recovery framework. The proposed design was implemented for a target SCADA network. In the preceding chapter the implementation was used to validate the design against the stated aim of this research. This section will continue the discussion by describing how the framework would fit into a larger recovery plan and presenting limitations of both the proposed framework and the technologies underlying the proof-of-concept implementation. It then offers future work in the field to both increase the utility of the recovery framework and increase SCADA network security overall. Following this discussion the work is concluded.

6.1 Discussion

This section includes a discussion of how this framework could be incorporated into a larger recovery plan, limitations of the framework, and potential future work. The noted limitations are drawbacks of the underlying technologies necessitated by the design or used by the proof-of-concept. Future work in the area could focus on ways of improving the capability of the framework in certain areas.

6.1.1 Trusted Recovery Framework in the Context of a Wider Recovery Plan

In section 3.1.2 some assumptions of the broader recovery plan in which the proposed framework was expected to be utilized by were outlined. This section will discuss those further in order to illustrate the sort of SCADA network recovery approaches that are made possible by the trusted recovery framework. While appropriate recovery approaches are dependant on the particular SCADA network and its operational environment, a general approach is outlined below in order to demonstrate how an approach might incorporate the proposed framework.

The proposed recovery framework provides the ability to simultaneously restore a group of nodes to a trustworthy state. The restored group of nodes can be relied upon to behave as expected, provided that some of the assumptions in 3.1.2 are adhered to. The group of nodes to be restored can range from a single node up to the entirety of the SCADA network and they will not be operational during the restoration process. This leads to a number of concerns, namely: how network capability is maintained during the recovery, how are restored nodes maintained in a trustworthy state following restoration, and what should be done if the verification phase fails.

An axiom of this research was that availability of the SCADA network is of the highest importance. Following a compromise, a SCADA network may be in a degraded but still operational state. As a result, operational capability may have to be temporarily sacrificed in order to conduct the recovery. While the appropriate recovery plan depends on the particulars of each individual SCADA network one approach could be to segregate the network into a compromised, recovered, and in process segments. This segregation policy could be implemented, for instance, with the use of Virtual Local Area Networks (VLANs). Following a compromise, the degraded network would continue to operate in the compromised segment. Certain nodes could then be transferred to the in process segment, along with the recovery server, where the restoration would occur. Once the restoration has been verified successfully the nodes could be moved to the recovered segment. The restoration of the final batch of nodes would involve a transfer of operational control to the recovered segment. SCADA networks with redundancy in nodes lend themselves to such an approach. This approach allows the SCADA network to remain operational during recovery and segregates the restored nodes with renewed trust from the compromised nodes. This second feature allows the trustworthiness of the restored nodes to be maintained following recovery.

The outcomes of the verification phase are enumerated in Table 3.3. Since each outcome is mapped to particular methods of malware persistence, the framework provides some information to the operator about the trustworthiness of a node following recovery. This information is one factor of a larger group which includes the role of the particular node in the SCADA network, the current operational context of the network, and any further threat information gathered during or prior to the compromise. Given all those factors a SCADA network operator must make a decision regarding the course of action following a failed verification. The framework cannot provide particular guidance, however it can help network operators better understand the risk of trusting a node following recovery.

6.1.2 Limitations

While this section may appear to focus on implementation specific limitations, such as those involving particular technologies used by the proof-of-concept, the choice of technology was often driven by decisions made by the design. For this reason, limitations in the underlying technologies are, in a large part, the result of the design. The framework's inability to measure any system firmware other than the BIOS has been identified and discussed in Section 3.3.3, therefore it is omitted from this section, however, potential solutions are discussed in the following section, Future Work.

Use of `ntfsclone` vs. `dd`

In the implementation `dd` is used to restore the MBR from an authorized image and `ntfsclone` is used to restore NTFS partition images. As described, `ntfsclone` writes only to the sectors which contain data used by the file system. As a result, when a partition is restored from an NTFS image, any data from the previous NTFS filesystem which resides outside of the blocks used by the new NTFS filesystem will remain on the disk following restoration. This represents stateful memory which is not measured as part of the verification phase. On the other hand `dd` writes out an image block by block, this overwrites the entire disk. Despite these risks, `ntfsclone` was chosen based on the timing constraints placed on the design. The performance of `dd` and `ntfsclone` was discussed in section 5.2. The performance considerations can be weighed against the increased risk of not measuring an aspect of the restored system.

Trustworthiness of the Intel TXT DRTM

Since the guarantee of integrity provided by the framework is entirely based upon the DRTM created by Intel TXT, it is worth examining the extent to which this root of trust can be relied upon.

There is a history of reported vulnerabilities for Intel TXT. In 2009 the Invisible Things Lab demonstrated methods of attacking Intel TXT using an SMM handler exploit [53] and the SINIT ACM [54]. In addition to SMM, the Intel ME can also bypass the DMA protections put in place by Intel TXT. Despite these reports we still decide to trust Intel TXT's DRTM. This is because TXT is still believed to add security value to a trusted recovery architecture by raising attacker costs considerably. So far, the only examples of attacks on Intel TXT have been laboratory proof-of-concepts. However, the fact that vulnerabilities are found in Intel firmware and extended instruction set re-enforces the importance of regular software and firmware patching.

Security Considerations of Enabled Intel AMT

Given the level of system access granted to Intel AMT it is an obvious security concern. Consequently, there has been a large amount of reporting on Intel AMT and potential vulnerabilities within it. As a result, enabling it within a SCADA network should not be done without due regard to security. However, the risk can be mitigated. Some of the reported 'backdoors' involved allowing an adversary access to a node which allows access to the BIOS configuration, leaving the default Intel AMT password in place, and allowing remote access to the the ports used by Intel AMT[55]. The attacker would then use Intel AMT functionality to remotely control a system. This sort of vector can be countered with basic protective measures. More noteably, in late 2017, Positive Technologies disclosed three vulnerabilities in the Intel ME firmware [56, 57, 58] which allowed arbitrary code execution. These vulnerabilities depended on four conditions: 1) AMT is enabled on the node 2) the AMT administrator password is known or can be bypassed 3) the BIOS password is known or can be bypassed and 4) the BIOS can be configured to allow write access to memory region assigned to the ME. Using Intel AMT as part of a Trusted Recovery Framework would necessarily satisfy the first condition and the fourth condition is dependant upon the BIOS implementation. Conditions two and three could be satisfied with insufficient security controls on the implementation of AMT or access to the BIOS password. Further to this, network controls can restrict access to the ME by remote systems and decrease the security threat.

Given this discussion, the security considerations of enabling Intel AMT are an important factor in the implementation of the design trusted recovery framework.

6.1.3 Future Work

There are several areas in which certain aspects of framework's approach could be developed further. These include progressing the use of a late-launch environment as a platform to measure firmware by including the capability to verify firmware other than the BIOS and increasing the scope in which the recovery process aids in forensic investigation.

Extended Firmware Measurement

One notable limitation was the framework's inability to measure firmware outside of the BIOS. As a result of this limitation, there is stateful memory on a node which is not measured as part of the verification phase. A potential improvement to the framework could include adding capabilities to measure other firmware, such as that of the NIC or disk drive. Projects such as `chipsec` and `viper` provide potential approaches to this problem [59, 60]. The late-launch environment provides a trustworthy environment from which the privileged operations required to verify system firmware can be conducted. An investigation into combining late launch techniques with advanced firmware measurement techniques would improve the capability of a trusted recovery framework. While verifying system component firmware increases the complexity of the framework, it adds considerable security benefit, particularly when advanced nation state level actors are included in the threat model.

Forensic Data Capture

This paper cited the importance of availability over other concerns as a reason to forgo forensic activities on a compromised node. As a result, the proposed framework actually destroys any potential evidence which may be on the disk during the recovery process. Future work could explore the idea of gathering evidence prior to the remediation activities. Such research would investigate methods for retrieving important artifacts from a compromised node prior to remediation. Such activity would aid post incident investigations, help determine the extent of the compromise, and, consequently, confirm that the compromise has been fully addressed by the remediation actions.

6.2 Conclusion

SCADA networks can never be entirely protected from computer security threats. As methods for detecting network attacks and the presence of malware mature, appropriate recovery processes must also be developed. The unique characteristics and operating environment of SCADA networks make recovery approaches used in IT networks non-applicable. This research utilized the ideas of integrity chaining [7] as well as using CPU-based late-launch to create an environment from which to conduct a remediation or system deployment [8, 3] in order to develop a design for a Trusted Recovery Framework which utilized security capabilities embedded in hardware. The resultant framework was capable of automatically restoring multiple nodes in a SCADA network to a trustworthy state following a compromise. This was validated with the development of a proof-of-concept implementation for IPMS, the SCADA system used onboard certain classes of vessel in the RCN. This research contributes knowledge to field of SCADA network recovery. As a result, future work into the detection of advanced firmware bootkits on SCADA networks and the incorporation of forensics evidence gathering into the recover process can be conducted.

6.3 Contributions

The contributions of this research are:

1. The enumeration of high level requirements for a SCADA recovery framework based on characteristics of SCADA networks ascribed within SCADA security literature.
2. The design of an architecture to support the trusted recovery of nodes on SCADA networks. This architecture leverages embedded security capabilities in modern computer systems for both a root-of-trust for the measurement and reporting of platform state during the recovery process and an out-of-band method of remote control to automate the process. The architecture also satisfies the unique constraints of SCADA network recovery.
3. A three stage method of verifying platform state during the remediation of a node as well as a discussion of the possible outcomes of these three stages and the implications on node security.
4. An implementation of the trusted recovery framework for IPMS which allows nodes to be verifiably restored to a trustworthy state and provides all associated operations required to support the restoration function.

References

[1] *TCG EFI Platform Specification For TPM Family 1.1 or 1.2*, Trusted Computing Group, 1 2014, revision 15.

[2] *Intel Trusted Execution Technology (Intel TXT) Software Development Guide*, Intel, 8 2016, revision 013.

[3] J. Schiffman, T. Moyer, T. Jaeger, and P. McDaniel, "Network-Based Root of Trust for Installation," *IEEE Security Privacy*, vol. 9, no. 1, pp. 40–48, Jan. 2011.

[4] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, "Guide to Industrial Control Systems (ICS) Security," National Institute of Standards and Technology, Tech. Rep. NIST SP 800-82r2, Jun. 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf (Accessed 2018-03-23).

[5] "Protecting Industrial Control Systems. Recommendations for Europe and Member States — ENISA." [Online]. Available: https://www.enisa.europa.eu/publications/protecting-industrial-control-systems.-recommendations-for-europe-and-member-states (Accessed 2018-05-02).

[6] X. Zhou, Z. Xu, L. Wang, and K. Chen, "What should we do? A structured review of SCADA system cyber security standards," in *2017 4th national Conference on Control, Decision and Information Technologies (CoDIT)*, Apr. 2017, pp. 0605–0614.

[7] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, May 1997, pp. 65–71.

[8] A. Regenscheid and K. Scarfone, "Recommendations of the National Institute of Standards and Technology," *NIST special publication*, vol. 800, p. 155, 2011.

[9] "Cybersecurity Framework Core (Excel)," 2014. [Online]. Available: https://www.nist.gov/document-3764

[10] B. Schneier, *Secrets and lies: digital security in a networked world ; [with new information about post-9/11 security]*, nachdr. ed. Indianapolis, Ind: Wiley, 2008, oCLC: 845002165.

[11] D. Kushner, "The Real Story of Stuxnet - IEEE Spectrum." [Online]. Available: https://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet (Accessed 2017-11-08).

[12] "How Similar Are WannaCry And Petya Ransomware?" [Online]. Available: https://www.forbes.com/sites/quora/2017/07/05/how-similar-are-wannacry-and-petya-ransomware/#556805ee46eb (Accessed 2017-11-08).

[13] "Developing an Industrial Control Systems Cybersecurity Incident Response Capability," p. 49, 2009.

[14] Knight, S and Leblanc, S, "When Not to Pull the Plug – The Need for Network Counter-Surveillance Operations," Tallinn, Estonia, Jun. 2009.

[15] North American Electrical Reliability Corporation, "CIP-009-6 — Cyber Security — Recovery Plans for BES Cyber Systems," Tech. Rep., May 2014.

[16] *Preboot Execution Environment (PXE) Specification*, Intel Corporation, 9 1999, version 2.1.

[17] *Unified Extensible Firmware face Specification*, Unified EFI Forum Inc., 5 2017, version 2.7.

[18] "Sans 660: Advanced Penetration Testing, Exploit Writing, and Ethical Hacking." [Online]. Available: https://www.sans.org/selfstudy/course/advanced-penetration-testing-exploits-ethical-hacking (Accessed 2017-11-08).

[19] A. Matrosov, E. Rodionov, and S. Bratus, *Rootkits and bootkits: reversing modern malware and next generation threats.* San Francisco: No Starch Press, Inc, 2018.

[20] tedhudek, "Kernel-Mode Code Signing Requirements." [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-requirements--windows-vista-and-later- (Accessed 2018-03-27).

[21] "Malware threats and mitigation strategies," Tech. Rep., 05 2005.

[22] “Mebromi: The First BIOS Rootkit in the Wild,” Sep. 2011. [Online]. Available: https://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/ (Accessed 2018-03-19).

[23] “Hacking Team Uses UEFI BIOS Rootkit to Keep RCS 9 Agent in Target Systems - TrendLabs Security Intelligence Blog,” Jul. 2015. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-uses-uefi-bios-rootkit-to-keep-rcs-9-agent-in-target-systems/ (Accessed 2018-03-19).

[24] “InsydeH2o® UEFI BIOS | Insyde Software.” [Online]. Available: https://www.insyde.com/products (Accessed 2018-03-27).

[25] “TPM Vulnerabilities to Power Analysis and An Exposed Exploit to Bitlocker,” Mar. 2015. [Online]. Available: https://theintercept.com/document/2015/03/10/tpm-vulnerabilities-power-analysis-exposed-exploit-bitlocker/ (Accessed 2018-03-28).

[26] R. J. Anderson, *Security Engineering*, 2nd ed. Wiley, 2008.

[27] B. Parno, J. M. McCune, and A. Perrig, “Bootstrapping Trust in Commodity Computers,” in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 414–429.

[28] J. Rutkowska, *Intel x86 considered harmful.* Oct, 2015. [Online]. Available: http://miscellaneous.archive.tjw.moe/x86_harmful.pdf (Accessed 2017-05-15).

[29] *TPM Main Part 2 TPM Structures*, Trusted Computing Group, 3 2011, level 2 Revision 116.

[30] “Dod instruction number 8500.01,” Tech. Rep., 3 2014.

[31] X. Ruan, *Platform embedded security technology revealed: safeguarding the future of computing with Intel embedded security and management engine.* Berkeley, California: Apress Open, 2014, oCLC: ocn892481808.

[32] B. Kauer, “OSLO: Improving the Security of Trusted Computing.” in *USENIX Security Symposium*, 2007, pp. 229–237.

[33] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog, “BIOS Chronomancy: Fixing the Core Root of Trust for Measurement,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516714 (Accessed 2018-05-02).

[34] "Trusted Boot." [Online]. Available: https://sourceforge.net/projects/tboot/ (Accessed 2018-04-24).

[35] W. T. Futral, *Intel trusted execution technology for server platforms: a guide to more secure datacenters*, ser. The expert's voice in security. Berkeley, CA: ApressOpen, 2013.

[36] M. Garrett, "Introducting ring -3 rootkits," Las Vegas, NV, USA, Jul. 2009. [Online]. Available: https://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf (Accessed 2017-10-24).

[37] A. Kumar, P. Goel, and Y. Saint-Hilare, *Active platform management demystified: unleashing the power of Intel vPro technology.* Hillsboro, Or.: Intel Press, 2009, oCLC: 648411394. [Online]. Available: http://www.books24x7.com/marc.asp?bookid=33062 (Accessed 2017-11-08).

[38] "Positive Technologies - learn and secure : Disabling Intel ME 11 via undocumented mode." [Online]. Available: http://blog.ptsecurity.com/2017/08/disabling-intel-me.html (Accessed 2017-11-08).

[39] "NVD - CVE-2017-5689." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5689 (Accessed 2017-11-08).

[40] Intel, "Creating Trust in the Cloud," p. 5, 2013.

[41] "OpenAttestation: Software Development Kit to enable remotely retrieval and verify target platforms integrity," Mar. 2018, original-date: 2012-03-30T06:08:14Z. [Online]. Available: https://github.com/OpenAttestation/OpenAttestation (Accessed 2018-03-27).

[42] E. Messmer, "Black Hat: Researcher claims hack of processor used to secure Xbox 360, other products," Feb. 2010. [Online]. Available: https://web.archive.org/web/20120130095246/http://www.networkworld.com/news/2010/020210-black-hat-processor-security.html (Accessed 2018-03-28).

[43] P. Maene, J. Gotzfried, R. d. Clercq, T. Muller, F. Freiling, and I. Verbauwhede, "Hardware-Based Trusted Computing Architectures for Isolation and Attestation," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2017.

[44] E. Gamma, *Design patterns elements of reusable object-oriented software.* Reading, Mass. [u.a.: Addison-Wesley, 1994, oCLC: 873714193.

[45] "TrouSerS - The open-source TCG Software Stack." [Online]. Available: http://trousers.sourceforge.net/ (Accessed 2018-05-02).

[46] "Intel® AMT High-level API | Intel® Software." [Online]. Available: https://software.intel.com/en-us/articles/intel-amt-high-level-api-intel-manageability-library-to-manageability-webpage (Accessed 2018-04-24).

[47] "Dnsmasq - network services for small networks." [Online]. Available: http://www.thekelleys.org.uk/dnsmasq/doc.html (Accessed 2018-04-24).

[48] "SQLite Home Page." [Online]. Available: https://www.sqlite.org/index.html (Accessed 2018-04-24).

[49] "Dapper: a simple object mapper for .Net," Apr. 2018, original-date: 2011-04-14T08:42:59Z. [Online]. Available: https://github.com/StackExchange/Dapper (Accessed 2018-04-24).

[50] "PXELINUX - Syslinux Wiki." [Online]. Available: https://www.syslinux.org/wiki/index.php?title=PXELINUX (Accessed 2018-05-02).

[51] (2017) Trusted boot. [Online]. Available: https://sourceforge.net/projects/tboot/ (Accessed 2017-10-3).

[52] "Buildroot - Making Embedded Linux Easy." [Online]. Available: https://buildroot.org/ (Accessed 2017-11-08).

[53] R. Wojtczuk and J. Rutkowska, "Attacking Intel® Trusted Execution Technology," p. 6.

[54] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Another Way to Circumvent Intel® Trusted Execution Technology," p. 8.

[55] "Backdoored in 30 Seconds: Attack Exploits Intel AMT Feature." [Online]. Available: https://www.bankinfosecurity.com/backdoored-in-30-seconds-attack-exploits-intel-amt-feature-a-10583 (Accessed 2018-05-03).

[56] "NVD - CVE-2017-5707." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-5707 (Accessed 2018-03-28).

[57] "CVE - CVE-2017-5706." [Online]. Available: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2017-5706 (Accessed 2018-03-28).

[58] "CVE - CVE-2017-5705." [Online]. Available: https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2017-5705 (Accessed 2018-03-28).

[59] "chipsec: Platform Security Assessment Framework," May 2018, original-date: 2014-03-06T21:57:10Z. [Online]. Available: https://github.com/chipsec/chipsec (Accessed 2018-05-01).

[60] Y. Li, J. M. McCune, and A. Perrig, "VIPER: verifying the integrity of PERipherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 3–16.