# MODELING THE SECURE BOOT PROTOCOL USING ACTOR NETWORK THEORY

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I AT MĀNOA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

DECEMBER 2017

By

Mark Nelson

Thesis Committee:

Peter-Michael Seidel, Chairperson
Dusco Pavlovic
Yingfei Dong

Keywords: Science of Safety, Formal Methods, Actor Networks, Secure Boot, Trusted Computing

Dedicated to:

Richard Leland Anshutz, 1930 – 2013

One hundred years from now,
It won't matter what car I drove,
What kind of house I lived in,
How much I had in my bank account,
Nor what my clothes looked like,
But, the world may be a little better
Because I was important in the life of a child.

## ACKNOWLEDGMENTS

I would like to thank my wife, Elizabeth, for enduring hours of ‘looking at the back of my head.’ This thesis would not have been possible without her loving support.

I would also like to thank my parents. I have rediscovered my family these last few years and I miss our time together.

I would like to thank my advisor, Dr. Seidel for his guidance and patience.

Drs. Yingfei Dong and Dusko Pavlovic brought the Scholarship for Service program alive here at the University at Hawaii. As my committee members, they were very kind during my ‘progress reports’. I am forever grateful for their support.

I would like to thank the staff and faculty of the Information and Computer Science department at the University of Hawaii: Janice, Victor, Gerald and Wes; Prof. Clark; Drs. Biagioni, Johnson, Lim, Robertson and Sitchinava. I learned so much! Thank you one and all!

To Gail: Thank you for being so good at what you do.

I’d like to thank the CyberCorps Scholarship For Service (SfS) program and the National Science Foundation for their financial support. I’d also like to extend my thanks to Dr. Ray Vaughn from the University of Alabama at Huntsville for getting us off to a good start here in Hawaii.

Finally, I would like to thank my son, Sam. For teaching me patience and forgiveness.

# ABSTRACT

We propose a framework for modeling and analyzing the security of cyber-physical systems, in particular the security properties of a Secure Boot protocol. By reviewing the history of safety in Aviation & Urban Development, we observe how their safety systems matured and identify key factors for their success such as economic incentives, investments in to obtain a deep understanding of a system's components, the ability to scale and the rigorous definition & analysis of objectives. Cyber-physical systems are hindered by the lack of rigorous models to express and analyze security objectives. Based on previous works of formal Actor Networks from Computer Science and informal Actor Network Theory from Sociology, we propose a mathematical framework to model cyber-physical protocols and analyze their security properties while keeping the interactions, in particular the physical interactions, flexible. Finally, the thesis develops four Actor Network models and analyzes security properties of a Secure Boot protocol.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1
# INTRODUCTION

Almost anything in software can be implemented, sold, and even used given enough determination. There is nothing a mere scientist can say that will stand against the flood of a hundred million dollars. But there is one quality that cannot be purchased in this way – and that is reliability. The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay.[23]

—C.A.R. Hoare, 1980 Turing Award Lecture

## 1.1 The Problem

As an industry, we have been developing software at scale for over 45 years[1], yet our ability to develop systems that are both user friendly *and* secure continue to plague us.

Why do we adopt protocols, algorithms and programs[2] that are nominally well-behaved[3] but are unsafe and insecure? Historically, the major emphasis in program development has been functional requirements and verification, while safety and security concerns to prevent undesired functionality has been an afterthought. With increasing safety and security threats, this emphasis in program development needed to change. Addressing software security is a significantly harder problem than ensuring traditional program correctness. The challenges of software security begin with the specification of security requirements and the establishment of practical security models. Without rigorous security models, providing any form of security guarantees for software and systems is elusive. In practice, the lack of security guarantees are replaced by the detection of harmful behavior and by the development of statistical risk models for (harmful) security attacks. In this environment software security and safety are treated in a similar way. Despite the shortcomings of the risk-based security of software and systems, the practical use of these systems for online banking and e-commerce has been thriving.

Recent security attacks try to circumvent detection by hiding in hardware devices and by gaining control before a system is fully initialized. For these attacks the detection of the security threats are becoming

[1]I've pegged the start of industrial-scale commercial/personal computing at January 1, 1970. This is the 32-bit UNIX time Epoch, Dennis Ritchie was developing the C programming language and the conditions were set for the mass commercialization of computing by companies like Intel, Microsoft and Apple.

[2]Henceforth, referred to simply as 'protocols' or 'programs'.

[3]They have the liveness property.

challenging and any risk model becomes harder to apply. These attacks have motivated the development of secure boot mechanisms and secure memory enclaves (like INTEL SGX) that can provide privacy from the overarching system. Both mechanisms rely on features of the processor hardware and physical interaction from the user. Their proper operation needs to be modeled as a cyber-physical system.

Our goal is to model and analyze the security of cyber-physical systems. Our analysis tries to shed more light into the security requirements of them and to showcase the application of a model for security properties of computer systems that rely on physical features. A broader application of these models could be used to improve the specification of software security requirements and lead to safer and more secure program development.

While most developers would *desire* to write secure programs/protocols, the available specification and modeling methodologies and tools, and the related economics are simply not in their favor. We motivate the economic aspects of secure system development based on historic perspectives regarding the attainment of safety in other industries.

This thesis extends Actor Network Theory (ANT), a framework that is uniquely suited to scale from the abstract system level to the gate level and interact with diverse participants. This framework can be used to model complex systems. Using Procedure Derivation Logic (PDL) we can formally express and reason about security properties of a system. This thesis takes inspiration from [16, 33] to extend models proposed in [40, 34]. We use ANT and PDL to analyze the Secure Boot Protocol.

## 1.2 The Contributions of this Thesis

The contributions of this thesis include:

1. A historical analysis of two industries: Airline and urban safety. This analysis yields two products: 1) Identifying rational economic models that make investments in safety. 2) An attempt to apply the historic lessons from these industries to cyber technology.

2. A refinement of the Actor Network modeling technique. Specifically, combining the spirit of sociological Actor Network Theory[32, 16] with the formal methods used in Actor Network procedures[40] and the notation of Procedure Derivation Logic in [34].

3. A methodology for modeling and analyzing security properties with Actor Network Theory.

4. The application of the proposed methodology to model and analyze the Secure Boot protocol.

## 1.3 What is Security?

Security becomes relevant when an asset needs to be protected against an adversary. For cyber-physical systems, an asset can be just about anything: Phones, digital resources, etc. The asset should be usable and have some value. The asset must be able to be secured by a principal[4] and the value of the asset should be commensurate with the cost of securing it.

A system's capabilities can be classified into one of four sets: What it can & should do, what it can do but shouldn't do[5] and what it can't do. See Figure 1.1. Both the safe operation of a system and preventing deliberate exploitation require avoiding the "What it *can* do and *should not* do" space. Designers should strive to minimize this set, eliminate it if possible, or as our historical analysis will show, seek to mitigate it[6].

Safety and security are very different concerns. Safety requires protection against random incidents. Security requires protection against intentional incursions[9]. As applied to Software Engineering, safety may be thought of as preventing the software from reaching certain conditions that it should not get to (bugs). Security, on the other hand, is ensuring a system behaves correctly even when deliberate attempts from an all-knowing adversary are made to circumvent control of an asset.

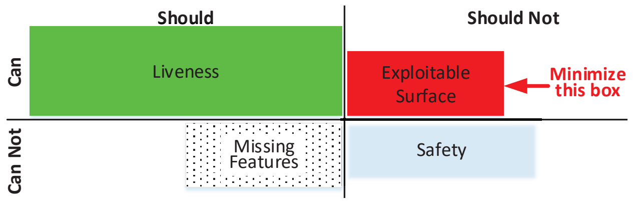


Figure 1.1: Classifying Capabilities into Sets

The analysis used in this thesis, like other formal approaches to security, does not distinguish chance from intent. The analysis only considers capability. Because of this, the following chapters will co-mingle safety and security[7]. The distinction between safety and security are still very important. Identifying attack models, scaling to the capabilities of an adversary and appreciating the difference between an accident and a choice are important. However, for the sake of simplifying this analysis, we will bypass intent and focus on capability.

System defects live in the gap between what the system should do and what it can do. Typically, these sets grow together. As a system adds useful functionality, the side-effects also tend to grow. It takes both time and cost to shrink the 'What it *can* do set'. Those same resources could be funneled into additional functionality or improving the usability of the system. Designers should carefully balance the investments of their resources, time and cost, into all three deliverables of a system: Usability, Functionality and Security.

The research in Chapter 2 will show that although there is short-term economic benefit to investing in usability and functionality (a local maxima), a superior long-term economic benefit exists (global maxima) when security investments are made to reduce the size of the 'what it *can* do' set.

### 1.3.1 Correctness vs. Security vs. Trust

[4]A decision maker or someone who controls an asset.

[5]For linguistic simplicity, however, we will call this what it *can* do.

[6]Many biological systems, such as the human body, have adapted to live with malevolent viruses and bacteria by minimizing their harmful effects rather than trying to eliminate the source.

[7]Norwegian only has one word for both safety and security: 'sikkerhet'[9].

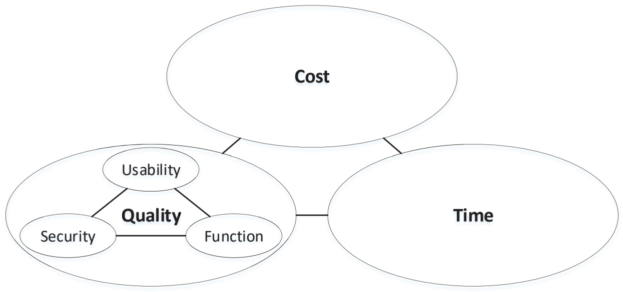


Figure 1.2: Balancing Conflicting Demands

In this thesis, trust is the idea that an input is 'trusted'; A guarantee that the input comes from a particular source that is trusted to 'do the right thing' (not introduce malware / has no malicious intentions towards the user of the software). This trust, should not be mistaken for correctness. Just because the software comes from Microsoft[8] does not guarantee that the software will behave correctly. Finally, even if the software is trusted and correct, there's no guarantee that it's secure.

TPMs and Intel TXT technologies provide load-time trust (the code is authentic at the time it is loaded), but it does not provide runtime protection. For example, the code may be changed after it is loaded or the program may contain bugs.

Readers should also be mindful of who is making the decision regarding trust. Oftentimes, one party will adjudicate (determine that X is trustworthy) trustworthiness and make an attestation to another party. Therefore it's also important to understand the reasoning behind why something is being trusted.

As an exercise in trust, a Windows computer running 'certlm.msc' lists hundreds of trusted parties in 'Trusted Root Certification Authorities \Certificates'. In this list, it is hard to identify who many of these trusted parties are, why they are trustworthy and to what extent. The assumptions of a typical computer user regarding the trust in this list is not based on intuition or reason[9].

Another consideration is *what* concepts are being trusted. Is the trust related to the code originating from source X? Is the trust related to concepts that X trusts? Even the communication with X might be a concern, i.e. is the channel with X trustworthy?

Practical security can not exist without trust. Without trust, interactions and sharing is impossible. A designer can build anything in isolation and security would not matter because it's isolated from all users. The key to practical security is to be cognizant of trust concepts and trust ratings and to consistently apply a process of adjudication and monitoring.

## 1.4 Quo Vadimus: Where Are We Going?

Why are we still producing insecure protocols? The high-tech industry is vibrant and growing. The vast majority of software engineers do not intend to write insecure systems.

There are several reasons for this disparity... one being that Computer Science is relatively young compared to Mathematics or Physics. This puts Computer Science at a distinct disadvantage relative to other sciences because the theoretical branches vice the applied branches are blurred in the case of Computer

[8]Picked at random from many other possibilities.

[9]How much would you risk on the trustworthiness of any given certificate? How much would you gamble that *every* certificate on that list is trustworthy?

Science where they are quite clear in the case of the other sciences. For example, the continuum from Theoretical Physics to Applied Physics to Engineering are relatively well understood. In Computer Science, however, the distinctions between the disciplines of algorithm/protocol design, language design, systems programming, software engineering and web development are not as clear.

This thesis observes that one reason we publish insecure protocols is because companies can get away with it. Publishing an insecure system does not carry enough of a penalty for rational people to choose to invest in security. Almost all software license agreements limit the liability of the vendor to the point that, other than reputational damage, there is no penalty for incorrect, insecure systems.

The solution is not to flog software engineers (or their companies) for every software defect. History tells us that one way forward is to:

1. Understand and formalize the objectives
2. Establish a set of standards
3. Penalize companies for noncompliance with those standards
4. Continually improve the standards

## 1.5 Orgainzation of this Thesis

Chapter 2 analyzes two industries and how they scaled from their early, developmental period into a mature, safe and reliable era. Chapter 3 includes a literature review of the technical aspects of this paper: Actor Network Theory and Secure Boot. Chapter 4 describes the methodology this thesis proposes for using Actor Network Theory to formally model protocols and provides a full example. Chapter 5 models aspects of the Secure Boot protocol. Finally, Chapter 6 concludes this thesis. Appendix A contains a glossary of terms and Appendix B contains a list of axioms used in this thesis but are defined in earlier works.

# CHAPTER 2
# A BRIEF STUDY OF THE SCIENCE OF SAFETY IN OTHER INDUSTRIES

Risks are the wholesale product of industrialization and are systematically intensified as they become global.[12]

—Ulrich Beck, *Risk Society*

History furnishes us with lessons for the present. Therefore, it may be worthwhile to study the history of safety and security in the Industrial Age to gain some insight into the Information Age. This brief history will study two industries: Aviation and electrical/fire/building safety. These industries have several things in common with high-technology such as:

1. They are a product of fundamental scientific research.
2. The research has been engineered for commercial and retail use for the general public.
3. The commodity has an inherent potential danger.
4. The commodity adds tremendous value to society.

The list is axiomatic, however item (3) may require additional explanation with respect to technology. While not all high-technology is inherently dangerous, the use cases where it *is* dangerous are easy to underestimate. Certainly terrorists who have been located through their cellphones have underestimated it. The average technology user makes certain assumptions about the privacy of the information that flows through that technology. Oftentimes these assumptions are not realized and data they assumed was safe and secure is, in fact, vulnerable to loss of privacy or tampering. This loss of security is an ever-present threat to users of technology which has led to financial losses, wasted time/efficiency (for example, restoring a computer attacked by ransomware), loss of privacy (eMails such as Sony and U.S. politicians) and loss of intellectual property (industrial espionage). Whenever we use technology, there are certain risks we are assuming, hence a potential danger, though not always to life and limb.

## 2.1 The Development of Aviation Safety

On December 17th, 1903, Orville Wright piloted the first powered airplane[1] and changed the world. This chapter is a study of aviation oversight and its evolution through the last century.

Figure 2.1 illustrates the FAA's 100-year track record of continuous safety improvement. Since 1954, the U.S. government has collected data on ”Passenger-Miles”[38, 39, 21]. If 20 people fly 1,000 miles in

[1]There is some debate as to who was the first to fly. Regardless of who was first, the industry took off in 1903.

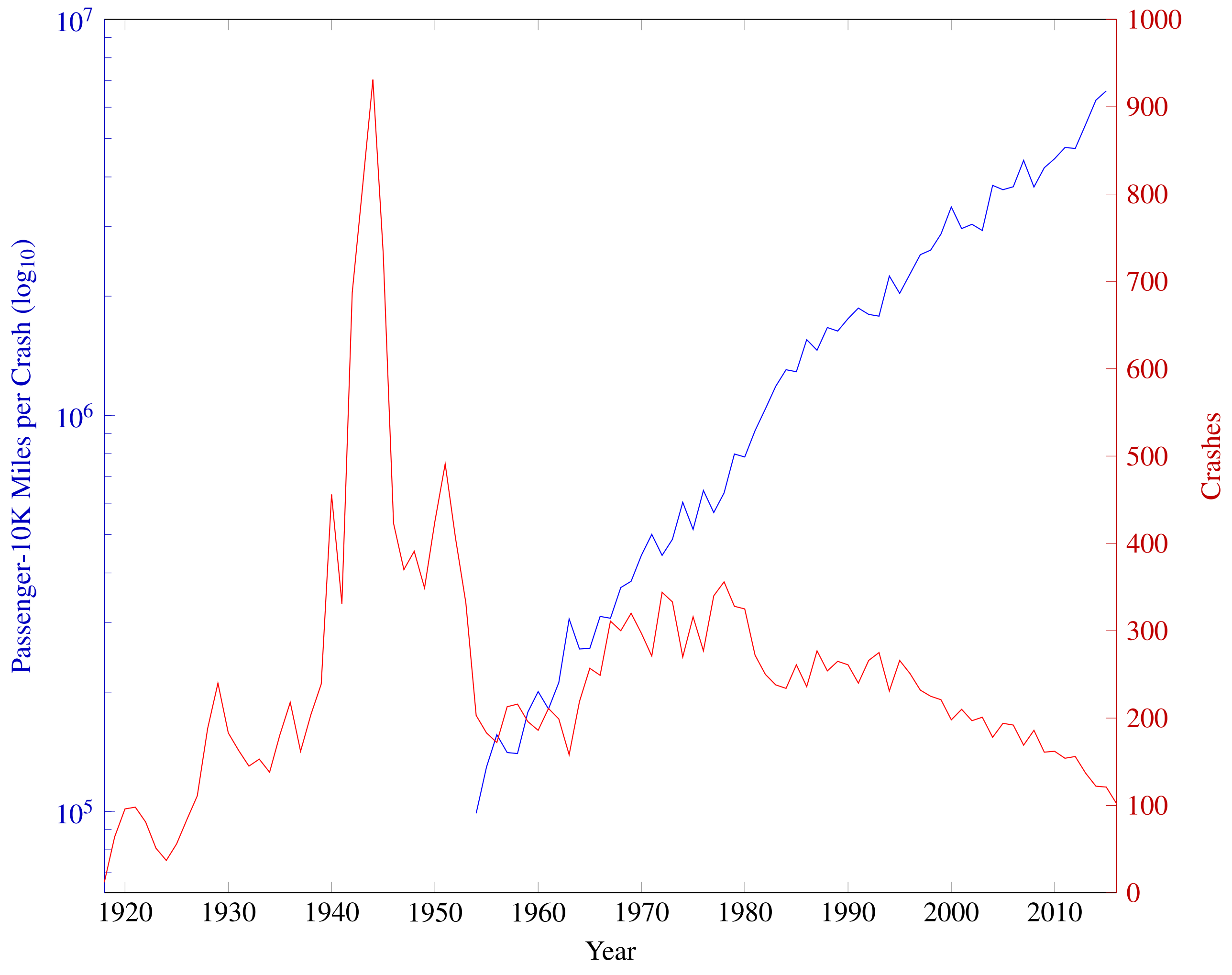


Figure 2.1: Record of Aviation Safety

one plane, then this flight is 20,000 Passenger-Miles. Worldwide crash data has also been compiled since 1918[36]. An examination of the crash data shows a relatively consistent rate[2]. What makes the FAA's success worthy of study is the declining crash rate given the exponential growth of passenger-miles. Note the left Y-axis in Figure 2.1 uses a $log_{10}$ scale while the right Y-axis is linear scale.

2016 was one of the safest years in aviation history. Since 1997, the average number of airline accidents has declined[41]. Clearly the FAA is doing something right, doing it consistently and keeping pace with the growth of air travel.

Section 2.1.1 is a brief review of the history of aviation safety. Section 2.1.2 summarizes axioms that worked for the aviation industry that could be applied to cyber technology.

[2]With the exception of WWII. Planes lost due to battle are not counted as a 'crash'. The uplift during the war years is due to the rapid expansion of the industrial base and cutting corners in safety to increase production.

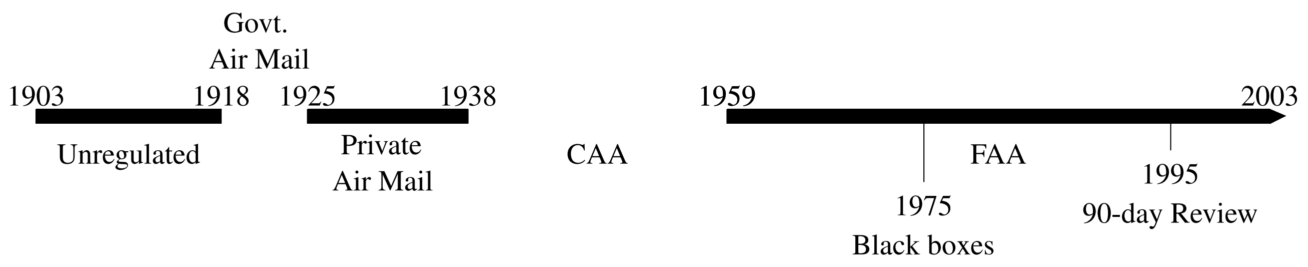


Figure 2.2: Timeline of Aviation Oversight

### 2.1.1 History of Aviation Safety

Aviation regulation has been tug-of-war between oversight and promotion punctuated by cycles of centralization and decentralization.

Early aircraft were notoriously dangerous. The first aviation fatality was Lieutenant Thomas Selfridge, who perished after a 1908 crash in a plane piloted by Orville Wright.

The first 22 years of aviation was unregulated[3]; however, the U.S. Government saw the utility of using airplanes to deliver mail and in 1918, it purchased its own airplanes and hired pilots as government employees. During this period, the government essentially operated its own airline. This was a period where two aviation systems ran in parallel: The U.S. Air Mail Service and an entrepreneurial private sector. The Air Mail Service operated a strict safety program with a 180-item checklist for every flight, medical exams for pilots and four mechanics for every aircraft.

Although the government's safety regimen required significant manpower, the nascent airline industries took note of their safety record. Commercial aviation had one fatal accident for every 13,500 miles flown. During the same period, the Air Mail service flew 463,000 miles per fatal accident. The U.S. Army had an even better record[30]. Unlike any of the other safety systems we will examine, federal safety oversight was requested by the aviation industry. This was due to the large social costs of accidents.

Commercial aviation needed to expand access to air travel from thrill seekers and wealthy travelers to a broad, 'rational' base of customers. The dangers of air travel created a reluctance for people to fly. Regulation was a proven mitigation for those dangers. Additionally, banks were unwilling to financially back airline industries until some "basic law" was created. Finally, insurance costs were prohibitively expensive as costs from irresponsible operators were distributed to responsible operations.

There were strategic reasons for government to 'invest' in aviation. If cities in a large country were linked by air, then citizens would be able to travel anywhere in the country in about a day. Any country able to achieve this goal would make it economically more efficient than its less-connected neighbors. Furthermore, the employment of pilots, mechanics and engineers for commercial purposes could be quickly converted to military production in the event of war.

[3]Unregulated in the United States. European nations had begun to regulate their airline industries and coordinate practices between nations through treaties.

The debate over the initial regulatory scheme spanned 6 years. Lawmakers had difficulty grappling with a new and growing technology, what types of controls to put in place and the balance between civil and military uses of the airways[4].

In 1925 Congress passed the Kelly Air Mail act that turned the U.S. Air Mail service over to private contractors. This required the government to 'certify' private companies using a scheme of inspection, licensing, data collection and enforcement[5]. The government needed to regulate the industry balancing unspecified economic boundaries; without excessive burden on the industry and without requiring a massive government workforce.

The 'type certificate' is a good example of that balance. The government would establish rules saying that an airplane needed to have A, B and C. The manufacturer would design aircraft meeting those requirements. The government would inspect the first aircraft and upon verification, issue airworthiness certificates to all aircraft built exactly the same way.

Another example of the government's attempts to scale were in medical examinations. Pilots needed to be certified fit to fly. The government developed the requirements and then licensed private practice physicians to certify pilots.

During this period, the government started to gather data and compile statistics that would eventually help regulators focus on classes of problems rather than individual accidents. Oftentimes, temporary certificates would be issued. Consistently, the accident rate of temporary certificate holders would be double that of those who had met all of the requirements. The regulation appeared to be directly linked to safety.

Oversight matured and expanded to airlines and flight schools. By 1935, twin engine certification required the plane to land on one engine and planes were required to have radios. There were limits on flight hours, dispatch procedures and airport controls such as lighting and employees.

In 1938, aviation oversight was reorganized into the Civil Aeronautics Authority (CAA)[6]. The goal was to clarify the government's role in three key areas:

- Promoting aviation. Setting policies to improve the efficiency of the system, setting rates, etc.
- Safety Regulation. Setting rules, managing the inspector workforce, engineering reviews and enforcement.
- Accident Investigation. Separating investigations into their own function served several purposes; The first was to de-politicize the process. Oftentimes, the investigation would recommend changes to regulation that created friction for promoting aviation. The second, and more important function, was to incorporate input from subject matter experts from private industry (for example, the engineers who helped design the plane). To get high-quality feedback, the investigation team needed to be separate from the enforcement mechanism.

[4]One of the reasons I find this topic so compelling is the similarities to the modern debate on the regulation of privacy and technology.

[5]Again, this is analogous to how ARPANET's governance transitioned to ICANN under the Department of Commerce.

[6]This is a digest of the history and is not intended to be a detailed historical account. This reorganization occured piecemeal in 1935, 1938 and 1939.

Aviation safety has progressively evolved from its early days to today. Although the CAA has reorganized into the FAA (Federal Aviation Administration), the core tenants are just as applicable today as they were during the agency's formative years.

### 2.1.2 Lessons from Aviation as Applied to Cyber Security

The initial U.S. Air Mail Service was an airline operated by the government. As such, the usual economics for a private enterprise did not apply. This allowed the Service to utilize a direct inspection and oversight regimen that required a tremendous amount of labor without the usual cost-benefit balance required in the private sector. This is similar to the origins of the computer. The first computers were built, or at least funded by, governments to achieve a goal that would have been unpalatable by the private sector at that time. The lessons are:

1. The government often funds the initial investments required to stabilize industries that have strategic and commercial applications.
2. Airports, harbors and ports have 'common' resources available to many users. To make these common areas safe, regulations and a safety system (such as lights and signage) improve its utility.

The aviation industry was unregulated for the first 22 years before both the public and industry demanded the government take action. The computer industry has been with us for just over 45 years and, as yet, the industry remains largely unregulated. One reason for this is the cost of failures are divergent. Every airline accident imposes a tremendous cost. The utility of computers in the first 25 years, from 1970 until the dawn of the Internet in the mid-90's, was large enough to invest in, but when these computer services were unavailable, the loss was not comparable to an airline crash. Today, the utility of high-tech devices is high enough that both the public and industry are starting to look for systematic (and hopefully scientific) methods to improve reliability. The lessons are:

3. An increase in utility drives a corresponding increase in losses associated with failure.
4. The public, industry and government must work together to balance utility with risk.

The airline industry took a long time to develop its regulatory structure, which evolved alongside the industry. The regulation required an ongoing partnership with the industry. And the government had contradictory roles to both promote the industry and enforce regulation. The government also had to involve itself in many diverse enterprises from material science to medicine. The lessons are:

5. The regulatory structure must scale. It is impractical for regulators to directly verify each of the safety requirements. The government's aviation regulators deployed several creative and economical tools to span the entire industry. These tools may be a template for improving the security practices of today's technology industry.
6. The regulation may spread to industries that are tangential to the industry being regulated.

7. The regulatory structure must adapt over time. It should be neither be too ambitious nor too shallow. It should cover what it can comfortably accommodate.

There are many unintended consequences to government regulation. It's expensive and slow. However, as this analysis will show, regulation is not *ipso facto* a bad thing. The airline regulators continue to measure utility and investigate mishaps. So long as the benefits outweigh the costs regulation can support an industry without killing innovation. The lessons are:

8. Useful regulation requires incentives that create a positive feedback loop that improves the balance of utility[7] vs. risk.
9. The regulatory structure should be data driven. Therefore, it must consistently measure utility over time as well as events such as failures.
10. The industry should carefully analyze failures, determine the root cause, evaluate existing mitigations and disseminate lessons learned.
11. The industry must understand the envelope of safe operations and operate within that envelope[8].

[7]Cost is factored into utility. Utility = (Benefit - Cost)

[8]This is not ment to stifle innovation, it implies that you test something before you use it and then apply lesson 7 to adapt the regulations.

## 2.2 The Development of Urban Safety

The electric lamp was patented in 1880 marking the end of the gaslight era and the beginning of the electric age[9]. By 1882, New York City had its first generating station and the commercialization of electricity was primed to spread around the world[27]. In a prescient article from the Cambridge Press about the loss of telegraph service and electricity after an 1888 winter storm:

> We have become so accustomed to the reception of news from all points of the world that we feel very sensibly even a brief deprivation, and we hardly know what to make of it if it lasts a day or two[2][10].

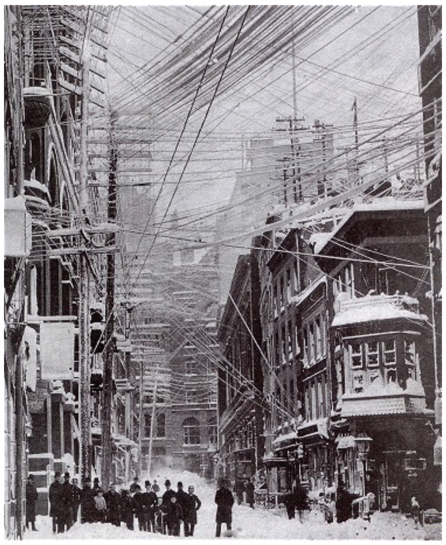


Figure 2.3: The Blizard of 1888

This section discusses three interdependent groups working together at the turn of the 20th century to apply scientific methodology to safety; Insurance companies, trade associations and governments allowed the nascent technologies of their era to scale.

Figure 2.4 identifies milestones in safety (top) and "Peak Losses" adjusted to 2015 dollars (bottom)[11]. The implementation of standards such as the National Electric Code (NEC) and the use of provably fireproof construction materials marked the peak of insurance losses for that era (The San Francisco Earthquake & Fire in 1906). The fact that 95 years passed before a larger loss was seen demonstrates the effectiveness of their approach.

**The Insurance Industry Spawns Trade Associations**

The study of electrical safety is intertwined with fire safety, both of which depend on a scientific analysis of building materials which was initially funded by the insurance industry. In the late 1800's, insurance companies had two predominant business models: Stock and Mutual[29]. The owners of a stock insurance company were stockholders and the economics that drove the company were based on maximizing the collection of premiums and minimizing payouts due to loss.

[9]On a personal note, this line of thinking came about as I toured two exhibitions at the Smithsonian Museum of American History: 'Lighting A Revolution: 19th Century Invention' and 'America on the Move'. The trip was funded through a government sponsored Scholarship for Service. I am grateful for the opportunity to visit the Smithsonian and bring something back to incorporate into this thesis.

[10]This tells us more about human nature than technology. Imagine the 'deprivation' modern society would feel if the Internet were unavailable for a day or two.

[11]The largest insurance loss is the 2001 attack on the World Trade Center complex in New York City.

[12]"Among the 20 deadliest fires in American history, 18 occured between 1865 and 1945.[29]"

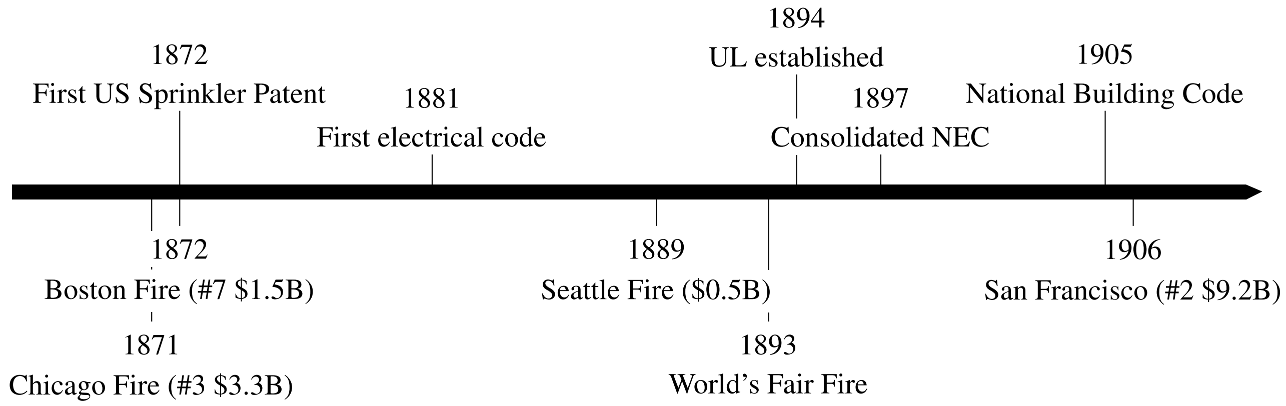


Figure 2.4: Peak Fire Losses at the Turn of the Century[12][7, 5, 37]

Mutual insurance companies, on the other hand, were owned by their own customers. Their economics were based on spreading risk and preventing mishaps.

> The mutuals philosophy resulted in the first efforts to innovate in fireproof construction on a scale that had previously been practiced only in very specific types of buildings, primarily courthouses and seats of government, hospitals, prisons and banks. [29]

The mutuals would set strict requirements for membership. Inspectors from the mutual insurance companies would make unannounced visits, conduct training, hold safety drills and bring back valuable risk information. The insurance companies also partnered with universities such as MIT and funded their own research because the inspectors were not trained scientists. Mutuals emerged as an alternative to the traditional insurance industry; midway between strong government regulation and the stock insurance model[29]. They achieved sustained fire safety with pressure on both customers and government.

Modern game theory explains the dynamics of a mutual insurance model. Companies, such as cotton mills, who were nominally competitors, would allow their experts to meet to discuss standards and exchange information on best practices. These rational actors saw value in co-operation rather than maximizing individual benefit (such as treating safety information as proprietary know-how).

In 1895, the presidents of five insurance companies met to develop a consistent standard for the employment of fire sprinklers. Their goal was to create a consolidated set of requirements for the manufacturers of sprinkler systems that would guide them in the development of an interoperable, modular set of products that could be sold to any factory that desired fire insurance. In 1986, this group would form the National Fire Protection Association or NFPA[5]. Their work would eventually produce the National Fire Code and the National Electric Code which established a clear set of rules for builders, governments and insurance companies.

Another insurance association, the National Board of Fire Underwriters or NBFU, was established in 1866. They funded two ambitious efforts: The publication of a National Building Code in 1905 and the financial backing for Underwriter Laboratories[13]. The first Building Code[14] was a 250 page document that covered everything from building materials to construction techniques to design (for example, increasing the number of exits).

The members of the NBFU were insurance companies, and as such, they moderated the influence of the laboratories scientists and the rule-of-thumb factory experts. The mutuals' interest was in results and they adopted the most pragmatic practice. The NBFU developed a methodology for improving safety[29]:

1. Isolate Risks
2. Convert them to technical problems
3. Solve the problems
4. Translate the solutions into new practices (materials, procedures or design)
5. Promote the new practices

The NBFU funded Underwriters Laboratories to be the "bench science" center of expertise. UL's mandate was to develop/publish requirements and test/certify materials and products. In 1906, they started the Label Service[15] to certify products. Subsequently, this became the most recognized symbol for product safety in the world.

Building codes needed to maintain a balance of affordability, usability and safety. The codes would often require materials to be certified to UL standard (this way the codes did not have to be rewritten whenever a new building material was introduced). However, they did not mandate 'fireproof' construction techniques. The authors accepted the fact that buildings will contain flammable materials and there will be fires. Therefore, they mandated the use of fire sprinklers and extinguishers to mitigate a fire by slowing how fast it can spread.

Ingenuity, cost and ease of use were key factors in the success of the standards. Many of the standards started with a debate between cost and safety. But eventually, as the technology matured and integrated with other safety systems, a quiet consensus would develop and the standard would be accepted. This "just works" invisibility was a key enabler.

There are significant parallels from the Industrial Age to the Information Age. At the turn of the century (1900), electricity had replaced a number of riskier technologies (such as oil lamps) to make society more efficient. Computers and the Internet significantly increase per-capita productivity. However, care should be taken to understand the risks associated with these gains.

[13]Founded as the Underwriters' Electrical Bureau and sometimes referred to as "UL"

[14]Actually, the first building ordanance is in the ancient Code of Hammurabi c 1754 BC: *Law 229, If a builder builds a house for someone, and does not construct it properly, and the house which he built falls in and kills its owner, then that builder shall be put to death.*[44]

[15]The first product to receive a UL label was a fire extinguisher.

The imbalance between known risks and perceived risks is highlighted in [12]. Ulrich Beck proposed that the same process of making the industrial metropolis[16] possible also manufactured a startlingly high level or risk. Beck proposed that this modern risk is self-perpetuating and irreversible. The knowledge of these risks is known to scientists but is invisible to the general public. He challenges the media and legal professions to discuss these risks openly and integrate them into our social and political landscape.

### 2.2.1 Lessons from Urban Safety as Applied to Cyber Security

The know-how to construct durable, fireproof buildings existed long before 1880, however, governments were the main practitioners of those methods[17]. It took major disasters to force insurance companies to push the practice into the private sector. This thesis does not propose that governments were the only entities with the know-how to build safely, only that their economics were aligned to build safely. For example, in the Fall of 1874, the NBFU called for a fire insurance boycott of Chicago to force local governments and builders to abandon their traditional building practices for safer materials and methods. The lesson is:

1. Regulators (governments and insurance companies) must balance cost vs. safety and pursue practical, "just works" solutions.
2. Safe construction requires both technical craft *and* economic advantage.

We need to rebalance product liability between consumers and producers of software. For example, one Microsoft End User License Agreement or EULA reads:

> NOTWITHSTANDING ANY DAMAGES THAT YOU MIGHT INCUR FOR ANY REASON WHATSOEVER ..., THE ENTIRE LIABILITY OF MICROSOFT ... SHALL BE LIMITED TO ... THE AMOUNT ACTUALLY PAID ... FOR THE SOFTWARE.[19][18]

The Uniform Commercial Code (UCC) is the legal framework in the United States that defines "Warranty Of Merchantability (Fitness For A Particular Purpose)" requiring manufacturers to deliver a product that is fit and safe for its intended purpose. The lesson is:

3. Our legal feedback loop is broken. Companies have created a system/product where they pay little to no penalty, other than consumer sentiment, for producing buggy (defective) software. The failures are not as catastrophic as an airliner crash or an industrial fire, but we are getting to the point where some systematic failures have a very large aggregate harm affecting millions of people and costing millions of dollars. Protocol failures have the potential to affect national or global economies in a way that previous technologies did not.

This thesis makes the following postulates:

[16]In the context of this thesis, the Internet is the modern equivilant.
[17]Courthouses, prisons and other government buildings
[18]... denotes language that was removed for the sake of brevity and clarity.

A. The expertise to publish secure protocols and software is available.
B. Although the High-Technology/Internet industry has a policy creation capability (such as the IETF), it lacks an effective enforcement mechanism.
C. **If software manufacturers must pay a proportional cost when they produce a defective product, then a beneficial ecosystem will develop, possibly driven by the software liability insurance industry.**

It's critical that standards bodies and regulators introduce practical guidance. It should be backed by multi-disciplinary scientific methods. It's also important to balance prevention with mitigation. A building fire is similar to a software defect. Both are going to happen, you just don't know where or when. Mitigation after you build it is expensive. The trick is to learn to live with it and contain its effects.

4. Disasters were substantially reduced as a result of disciplinary creativity and a tight fit between the experts and methods of policy creation[29].
5. Expertise was drawn from academics in university science and engineering departments, firefighters, architects, city planners, journalists and consumer advocates.
6. Fire, building and electrical safety started from privately-funded endeavors that drafted the practices governments eventually mandated.
7. With any new technology, generalization and abstractions are not easy to identify. Failures must be carefully studied to identify discernible patterns.
8. In 1996, the NFPA produced almost 300 codes and standards developed by more than 205 technical committees staffed by more than 5,000 volunteers[3]. This compares to the results of the Internet Engineering Task Force or IETF which has over 8,000 Requests for Comment (RFCs) and nearly 20,000 authors[6]. The key difference between these standards bodies is that the NEC carries the force of law in most jurisdictions in the United States and the IETF does not.

Getting the public to understand that safety and security comes at a cost that, ultimately, has value to them takes some effort. Safety associations spent a significant effort in public outreach, education and value creation. The insurance inspectors brought back information and distributed it.

9. *Users* of technology, not just developers, should learn about the benefits of security. Trade associations should highlight failures, discuss the science of security, mitigations and promote the idea that the cost of enforcing a secure technology ecosystem has value to them.
10. Insurance inspectors for software liability insurance companies should be able to distribute anonymized best practices and lessons learned publicly to any interested parties. Although the press has an important role in this process, we should attempt to replace 'investigative journalism' with informative and educational information.

This thesis makes the following postulate:

D. Today's obsession with privacy and being first to know something is eroding the pooling of knowledge necessary to distribute risk.

Although we would like to believe that our high-tech community is a modern and efficient ecosystem, in a historical light, it may be that our technological underpinnings are the equivalent of the wooden firetraps of the late 1800s. Technology is evolving at a rapid pace as we make improvements on top of improvements. At the same time, we do not yet have a deep understanding of these 'building materials' (re. Operating Systems, Libraries, Biometric system X, etc.). Our risks have aggregated to the point that a technological collapse would have major effects in the physical world. The lessons are:

11. Our technological underpinnings are brittle. Although they are functional, they are complex and not always well understood – especially when integrated with other technologies.
12. It took 80 years for fire sprinklers to become pervasive[3]. It takes a lot of time and, unfortunately, a lot of loss before the economic incentive to spend money on a mitigation strategy is proven and becomes mainstream.

This thesis makes the following postulate:

E. A society that can work together to produce better technological systems will be more economically efficient that their cyber-peers.
F. Developing an economically practical science of software and protocol security is imperative and it will power an engine that will produce benefits for generations.

Urban safety is primarily based on construction. Reacting to fire dangers is too late as it is too expensive to retrofit buildings with fire-safe materials. Therefore, the emphasis for urban safety is on construction. Aircraft safety model, on the other hand, is based on both construction *and operations*.

The common thread of these safety methodologies is the application of formal methods to reason about safety. These methodologies took a scientific, observational approach to identify root causes.

There is some good news, the Association of Computing Machinery (ACM) has recently published its first draft of Curriculum Guidelines for Undergraduate Degree Programs in Cybersecurity[8]. ACM is the same organization that codified the curriculum for Computer Science in 1968.

> "Cyber Sciences" reflects a collection of computing-based disciplines involving technology, people, and processes aligned in a way to enable "assured operations" in the presence of risks and adversaries. It involves the creation, operation, analysis, and testing of secure computer systems (including network and communication systems) as well as the study of how to employ operations, reasonable risk taking, and risk mitigations. The concept of 'Cyber Sciences' refers to a broad collection of such programs, and disciplines under this umbrella often also include aspects of law, policy, human factors, ethics, risk management, and other topics directly related to the success of the activities and operations dependent on such systems, many times in the context of an adversary.[8]

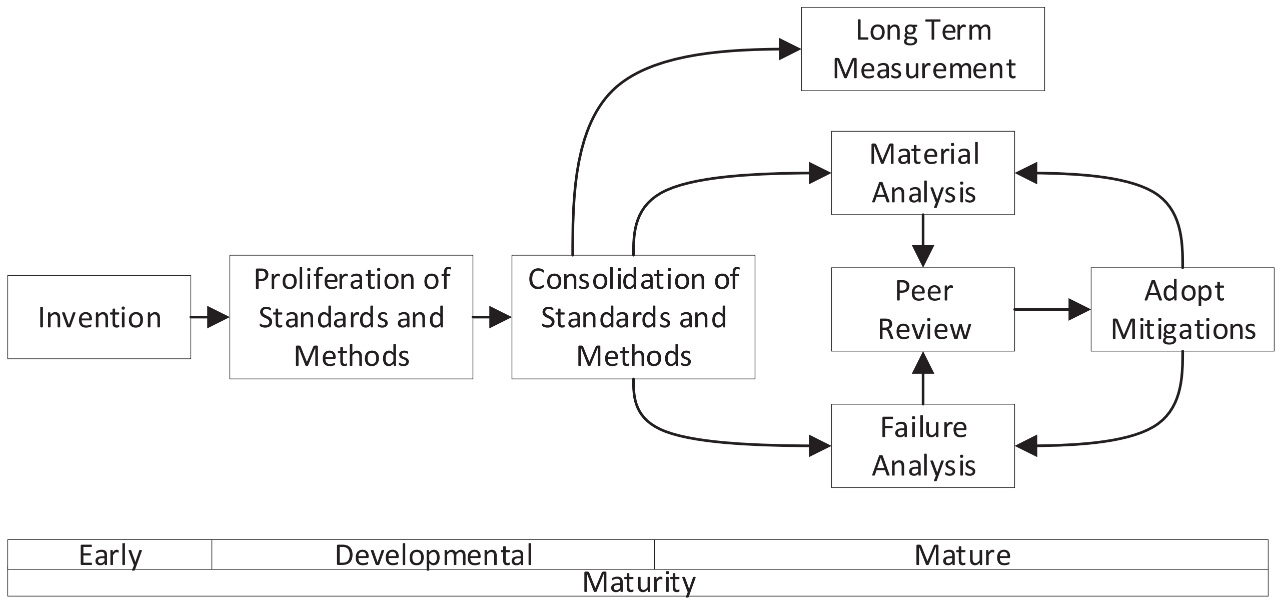


Figure 2.5: Technology Maturity Model

Clearly, cyber security is emerging as an identifiable discipline. Although this is a step in the right direction, programs founded on this curriculum likely generate practitioners and not designers of new protocols. In other words, the development of new formal methods of verification is probably outside the scope of Cyber Sciences.

There are some prescriptive, outcome-oriented standards such as PCI and HIPAA, but we have not, as an industry, achieved a mature level of safety and security. Perhaps this is due to the infancy of the fundamental science. Perhaps it's commercial interests. Or, maybe it's just too hard.

# CHAPTER 3
# BACKGROUND & RELATED WORK

If laboratories and research sites are to the twentieth century what monasteries were to the twelfth, then the sources of their power and efficacy remain a mystery.[16]

–Michel Callon, *Mapping the Dynamics of Science and Technology*[16]

This thesis uses Actor Network Theory to analyze the security properties of the Secure Boot protocol.

## 3.1 Actor Network Theory

In the 1980's, a group of European sociologists were dissatisfied with the direction of their field. The tools available to sociologists were limited; people, politics and relationships do not routinely lend themselves to many scientific processes. Furthermore, the group felt that sociologists had misappropriated a few scientific tools (such as induction[33]). To bring scientific methods back to sociology, this group began a line of research studying science, scientists and technology[16, 31][1].

During this period, technology had reached an inflection point and advanced at unprecedented rates. Technology's influence on society was profound[43]. A significant challenge for sociologists at the time was the complexity of the things they were attempting to study. They did not have *scientific* tools to describe interactions between complex people and complex technology.

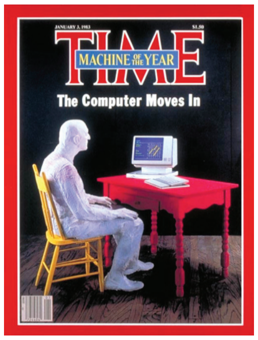


Figure 3.1: A 1983 Cover of Time Magazine

One of the first tools the team postulated was the 'Black Box': "When many elements are made to act as one.[31]" They used it as a tool to manage complexity[2]. It's not that they invented abstraction, but they did critically analyze it.

The term "Actor-Network" first appears in [16]. The original postulation of Actor-Networks is a metaphysical discussion of complexity and Heisenberg-ish observation that is not applicable to Computer Science. However, two themes emerge that are directly applicable to the Actor Network Theory proposed in [40]. The first is the idea that 'black boxes' are themselves composed of many 'black boxes'. Again, the European team applied a scientific rigor to these ideas that was novel at the time. The second, somewhat controversial theme, was the idea of homogeniety.

[1]*Science in Action* makes extensive references to Tracy Kidder's 1981 book *The Soul of a New Machine*. This book was required reading in my first class in Computer Science, Mike Dobeck's Assembly Language course at Sierra College in California.

[2]The term had been used for convenience since the '40s. The European sociologists defined it and used it scientifically.

In Actor-Network theory, all Actors[3] are first-class citizens. Presidents, babies, keyboards, elevator cabs, electric vehicles and cellphones are all on a level playing field. This makes it an effective tool for modeling the interactions between people and technology. However, the approach is not without controversy as ANTs do not consider human factors such as belief, preference, bias or rebellion. Fortunately, this is not a limitation for our purposes.

In [40], the authors adopt the spirit (and name) of [32]'s work; however, [40] is unique and is the starting point for this thesis. I'll use [40]'s own words to describe the relationship between [40] and [32]:

> The idea that people, computers, and objects are equal actors in such networks imposed itself on us, through the need for a usable formal model, even before we had heard of the sociological actor-network theory. After we heard of it, we took the liberty of adopting the name actor-network for a crucial component of our mathematical model, since it conveniently captures many relevant ideas. While the originators of actor-network theory never proposed a formal model, we believe that the tasks, methods and logics that we propose are not alien to the spirit of their theory.[40]

When necessary, we distinguish between [32]'s Actor-Network Theory and [40]'s Actor-Network Theory. On those occasions I'll denote them as Actor-Network Theory[32] or Actor-Network Theory[40].

## 3.2 Secure Boot

Computers can't do much without software. Typically, the very first program a computer runs is a piece of 'firmware' called the BIOS (Basic Input/Output System). It is used to start the computer, initialize devices and eventually, handoff the thread of execution to a system loader.

The evolution from BIOS to UEFI (Unified Extensible Firmware Interface) is still in progress. Standards organizations such as the Trusted Computing Group (TCG), the UEFI Forum and the National Institute of Science and Technology (NIST) have collaborated with industry partners to define a common set of implementation guidelines for how secure chains of trust should be implemented.

NIST has developed recommendations for BIOS Integrity and Measurements [42, 17] intended to help establish a secure measurement and reporting chain. However, BIOS vendors are free to implement and market 'Secure Boot' as they see fit.

Originally developed by Intel, the UEFI Specification [49] and the UEFI Platform Initialization Specification describe a framework that BIOS manufacturers, hardware designers and operating system vendors can use to securely boot a system.

The TCG encourages vendor-neutral collaboration and publishes an extensive set of specifications for a broad spectrum of technologies. One of TCG's working groups manages the Trusted Platform Module (TPM) specifications [47, 46, 48] that describe the API (application program interface) and use models for

[3]Latour prefers the term 'Actant'.

critical phases in the secure boot process. TPMs were not part of the original PC-AT design. A commonly accepted TPM specification [10] was published in 2003 and by 2009, TPMs had reached critical mass in that they were broadly available on most PC platforms [50], and vendors could rely on their presence. TPMs have also been adopted in mobile and networking devices.

TPM vendors such as Atmel [11], Infineon [24] and Intel [25] publish application notes documenting recommended procedures and details to help designers secure their systems.

While these references describe established and commercially used methods, protocols and responsibilities for implementing trusted boot, there is a growing number of studies that illustrate various shortcomings of implementations or weaknesses within the specifications themselves.

BIOS implementation flaws in laptops are described in [15]. In [52], the authors describe weaknesses in the BIOS update process such that if malware were to gain kernel access, undetectable code could be written to flash memory. In 2013, procedures for bypassing Windows 8 Secure Boot by writing into NVRAM and disabling trusted boot have been found [14]. In 2014, overflow vulnerabilities in UEFI firmware were discovered that allowed application software (helped by Windows 8) to introduce malware as a pre-boot driver [28].

There are several lines of research and technological fronts being advanced simultaneously. The term Secure Boot typically refers to a feature of UEFI that ensures the firmware that gets a computer ready to launch an operating system. The term Trusted Boot refers to the next stage in the boot process, measuring the trustworthiness of the operating system or hypervisor and ensuring that all of the software in those environments are trusted. The term Trusted Computing generally refers to using applications using TPMs to store credentials or other information. For example, using a TPM to store BitLocker keys or verify Digital Rights Management protocols are examples of Trusted Computing. In this thesis, we bundle the terms Secure Boot and Trusted Boot together and simply refer to them as Secure Boot.

# CHAPTER 4
# METHODOLOGY

It is not enough to do your best; you must know what to do, and then do your best.[1]

—W. Edwards Deming

It's important to carefully bound our claims. Our intention is to analyze an existing, complex protocol[1] and attempt to discover things about it that may not have been apparent. We will not present a formal verification of the protocol – building up axioms to claim the protocol is or is not secure. An axiomatic approach might be used if an author intends to present a new 'secure' protocol and would use ANT to prove its correctness and is left for future work.

This section is presented in the following order: 1) Discuss the intuition of Actor Network Theory. 2) Define the terminology used. 3) Demonstrate the notation. 4) Briefly identify the axioms used in these models. 5) Outline a proposed methodology for Actor Network modeling. 6) Provide an example of a complete Actor Network model.

## 4.1 The Intuition of Actor Network Theory

Sociological Actor Network Theory[32] allows researchers to study complex systems using scientific tools. Let's say Alice steals \$100 from Bob. Bob has several options, such as stealing the \$100 back or fighting Alice. Another option is for Bob to avail himself of the legal system[2]. Bob is giving up some options in the hopes that a judge will arbitrate a fair resolution. A model of these social interactions could be built. The model may contain *types* such as People (who can possess money), actors such as Alice, the \$100 and a courtroom (at a minimum, Bob and the judge should be together in a courtroom (*a configuration of actors*) to arbitrate the case). Courtroom etiquette demands prescribed channels of data flow. Principles such as Alice and Bob can control the actions of other actors such as moving the \$100 around. In this case, actors, operating within their defined roles, move through a variety of configurations working towards a goal. Upon completion of the goal, the actors are free to go about their business.

Actor Network Theory[32] gives sociologists scientific tools to, for example, apply game theory to a legal system vs. a natural-law system (where Bob strong-arms Alice to get the \$100 back). Or to study the economic efficiencies of societies that use different models to arbitrate disputes. One innovation of Actor Network Theory[32] is the notion of the 'Black Box' – a method of abstracting complexity by nesting Actants within Actor Worlds to simplify complex interactions between people and their environment.

[1]It is not strictly a protocol. Secure Boot is an implementation of a set of technologies. I have endeavored to backtrack through standards, app notes and the like to cite and document the implementation and analyze it as a complete protocol.

[2]In this example, the legal system is a protocol.

In Actor Network Theory[40], a model is developed called the Actor Network. The Actor Network maps elements from the protocol to nodes in the model. Furthermore, it connects the nodes using channels. After the Actor Network model is defined, Procedure Derivation Logic (PDL) is used to: 1) Define Hoare triples (a set of 'before' states) + (a set of moves) + (a set of 'after' states), 2) Define the goals for the protocol and 3) Apply logical tests to determine if the goals were met (or not).

This thesis draws its inspiration of Actor Network Theory from [32] and [16]. It uses [40] as the source for the modeling techniques' theoretical underpinnings of ANTs. Finally, it uses Procedure Derivation Logic from [34] as the starting point for this thesis.

We want to apply Actor Network Theory to model Cyber-Physical systems, a discussion of it is in order. A Cyber-Physical system can be almost anything. Civilization has had physical systems for ages and while cyber systems are relatively new, I'd posit that a cyber system's value is marginal if it doesn't, in some way, effect a physical system[3].

Cyber-Physical systems depend on protocols to accomplish their goals. "to uncover the mechanisms of power of science and technology, it is therefore important to reveal the ways in which laboratories simultaneously rebuild and link the social and natural contexts upon which they act.[16]" Protocols glue these systems-of-systems together. **The goal of Actor Network Theory is to use scientific methods to model protocols and analyze their performance against goals.**

ANTs and PDL do not *require* a threat or adversary model because PDL uses axioms to evaluate its security claims. That said, a threat model could be added for a more focused analysis.

## 4.2 Terminology[4]

The spirit of the original Actor Network Theory[32] has valuable insight for modelers. There's a lot of reasoning that went into the theory that applies to this proposed methodology. However, for the sake of clarity, it is also constructive to employ a concise set of terms that are both descriptive and practical. Table 4.1 is a comparison of the original terminology used by Bruno Latour & Michael Callon in [32, 16] with the terminology used by Catherine Meadows & Dusko Pavlovic in [40, 34]. Table 4.2 defines the terms used by this thesis' proposed methodology. Wherever practicable, the definitions in Table 4.2 are extensions of the definitions in Table 4.1.

In addition to the definitions in Table 4.2, there are a few concepts that need further elaboration.

Actors may hold a state. This state may advance based on internal processes or due to stimuli from the network[6]. While individual actors autonomously control their own behavior, the protocol designates the

[3]An intersting topic, but for another paper.

[4]This section defines terms specific to Actor-Network Theory. Appendix A contains an expanded glossary of acronyms used throughout the paper.

[5]In [34] the term is Box Configuration articulated in Definitions 2 and 3.

[6]An actor receiving a stimuli does not ipso facto know which actor sent the stimuli. The source actor places a message on the network which delivers it to another actor. The receiving actor only knows that a message came from the network. It requires additional information (via the protocol) to prove it's receiving a message from a known/trusted source and that it has not been tampered with.

Table 4.1: Terminology Used in Prior Works

| **Callon & Latour** | | **Meadows & Pavlovic** | |
|---|---|---|---|
| **Term** | **Definition** [32, 16] | **Term** | **Definition** [40, 34] |
| Social Interaction | | Protocol | An orchestration of actors executing actions to move through states to achieve a cooperative task. |
| Actant | Members of different or overlapping worlds. | Actor, Node | A computational agent who participates in a configuration. |
| Actor World (AW) | Heterogeneous actors that come together. Without one actant, the AW would be something else. AWs associate heterogeneous entities. | Configuration | A mechanism, assembled from separate components that may be owned or controlled by different principals. |
| Actor Network (AN) | The interconnected lines between AWs. The terms AN & AW draw attention to different aspects of the same phenomena. AW emphasizes unity and self-sufficicency. AN emphasizes structure and change. The terms are used interchangeably. (See Figure 4.1) | Actor Network | Actor-networks depict social processes as computations. Configurations of nodes nested in higher-order configurations. |
| Translator Spokesman | The spokesman of the entities he constitutes. | Identities Principals | Principals control actors. An principle's identity is a unique actor (like a password or private key) controlled solely by the principal. |
| Translation Geography | A geography of necessary points of passage. A strategic point through which an AW must pass. | Channel | Communication channel between agents. Unless noted, there is no guarantee that the channel is secure, trusted or authentic. |
| Translation Displacement | The actual messages carried across channels. They may be reports, memorandum, documents, survey results (they are sociologists after all) and scientific papers. | Messages, Events & Actions[40] Move[34] | Computation in a network consists of events. When a principal initiates a message, its an action. When a node receives a message, its an event. |
| | | Predicate | Identifies the state of an actor. |
| | | Role & Network Procedure | Actors play a role assigned to it by a network procedure. |
| | | Channel Type | Not all channels are the same. For example, Cyber Channels have no notion of distance and the recipient can not observe the sender. A visual channel requires some level of proximity. Binary channels stream bits from one node to another with some level of stochastic reliability. |

Table 4.2: Terminology Used in this Thesis

| **Term** | **Definition** | **Identification** |
|---|---|---|
| Protocol | Should provide a benefit to principals. | None |
| Actor | An entity in the model that plays some role in the protocol. Cyber, human and physical actors are all first-class citizens. Actors can be nothing, a single item or a network of items. Actors may encapsulate a complex machine at a particular state. | A descriptive label, italicized, camelCase w/ lowercase first letter, e.g. $bus, cpu_1, tpm$. |
| Principal | Principals are a special type of actor that control other actors. Principles are the ultimate beneficiaries of a protocol. They may be identified by a unique, atomic element that is a known, possessed or a measurable property. | Either capitalized first letter or ALL CAPS. Descriptive italic label, e.g. $CPU$, $People$ |
| Type | A set of actors that share a common characteristic. When types are used, zero, one or more members of that type could be substituted (unless otherwise constrained). For example the type $\boldsymbol{people}$ could be {Alice, Bob}. | The same nomenclature of its members in bold. e.g. $\boldsymbol{cpu}, \boldsymbol{User}$ |
| Configuration | A hierarchical set of Actors. No two actors in the hierarchy[5]are the same. | Due to the nature of Actor Network Theory (where actors are black boxes consisting of other actors), the notation is the same as an actor. Configuration names may follow { } as a subscript. e.g. $\{a1, a2\}_{configName}$. |
| Action | Actions are initiated by Principles and may be executed by actors. An action should change the state of the network and is denoted by a Hoare triple: Prior state + Action + Post state. An action could be sending or receiving a message. | Use square brackets to name an action, e.g. $[saveData]$, $[driveBus]$ |
| Policy | A logical statement that defines a set of legal transactions and/or configurations. Should be a subset of all possible actions ∪ configurations. | A person can only board the bus if it has an empty seat, e.g. $(seat == emptySeat) \Rrightarrow (\boldsymbol{person}) : [boardBus]$ |
| Goal | The goal of a protocol should be defined in both plain language and in mathematical expressions that should be true (or hold true) if the goals are met and false for all other cases. | The Bus should earn money, e.g. $(paidFare \sqsubset Bus)$ |

Table 4.3: Keywords Used in this Methodology

| **Keyword** | **Description** |
|---|---|
| IDENTIFY | Assign a label *and* a meaning to something. |
| ENUMERATE | Identify all possible members for an actor or type. Consider whether those members are mutable or immutable[7]. |
| ASSIGN | Set configurations. |
| DESIGNATE | Define an allowed action or policy. |
| TEST | Ensure a given rule holds for a given set of configurations. |

messages and sequence of tasks the actors *should* perform.

*Well-Behaved Actors* are predictable and execute tasks according to protocols that determine their interaction with other actors through available transactions. *Mis-Behaved Actors* are unpredictable and possibly malicious. Regardless of their 'motive', these actors do not always function within the bounds defined by the protocol. A protocol analysis must be blind to the intention of actors and focus on what actors *can* or *can not* do. On the other hand, the goals of the protocol should align closely with the intentions of the actors.

When things are true they hold
When things hold they start becoming true

Figure 4.1: The Duality of Actor Worlds & Actor Networks

This thesis does not use Channels although the concept is articulated in both [16, 40]. It should be noted that [34] does not make significant use of channels. From a practical perspective, channels are diagrammatically useful, but the ephemeral nature of 'allowed moves' makes diagrams with channels difficult to interpret or too numerous to be helpful.

Our methodology uses the keywords in Table 4.3 to denote specific actions. For example IDENTIFY instructs the modeler to assign a label to an actor *and* document a definition for that label. The inspiration of these keywords is RFC 2119's definition of MUST and SHOULD.

A *Conversation* or *View* represents the Protocol as experienced by any given actor. Although this methodology does not utilize conversations (because they do not alter the logical expressions of the model) they can be helpful for visualizing how a protocol may be used. From an analysis perspective, however, conversations of two communicating actors should agree based on the policies of the protocol.

[7]At the time of this writing, I do not have a good method for identifying the cardinality of this membership, which I think is an important part of the model. Cardinality should denote: Choose-zero-or-one, choose-one, choose-one-or-more, Coose-*n*, etc.

Table 4.4: Distinguishing Includes from Contains

| **Left Term** | **Operator** | **Right Term** |
|---|---|---|
| element | $\in$ | set |
| set | $\sqsubseteq$ | set |

## 4.3 Notation

Table 4.5 describes the mathematical operators and terms used in this thesis.

As this methodology is intended to be used for analysis, labels should be descriptive. Instead of $a$ or $\alpha$ this thesis will endeavor to use descriptive labels such as *Alice*, $emptySeat$ or $\boldsymbol{person}$. The labels will use the camelCaps method of combining words into a single label. The members of a configuration should be distinct and easily identifiable. In our examples, several actors have a *pouch* so they may be identified as $pouch_{Alice}$ for Alice's pouch and $pouch_{Bus}$ for the Bus' pouch.

The logical tests $\in$ and $\sqsubseteq$ are similar but distinct. In casual usage the words "includes" and "contains" are synonymous, but in logic they have distinct meanings[4]. Table 4.4 illustrates the differences in their use. In Actor Network Theory, we would could say € $\in$ *Alice* if Alice has some money. We could also say that $\{€\} \sqsubset Alice$ if we wanted to say that Alice has the € configuration.

To reduce ambiguity in the oft abused = sign, this thesis will avoid its use where practicable and use four distinct operators: == TESTs for equality, := makes an assignment, ::= and $\ltimes$ define membership in configurations and types respectively.

Before ASSIGNing a value to a configuration, modelers should define (or ENUMERATE) its configuration. The next two sections discuss the notation for ASSIGNing values and ENUMERATING configurations and types.

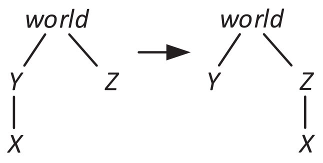


Figure 4.2: Moving X

### 4.3.1 Configuration Notation

An ENUMERATEd configuration identifies two things: Permanent members and potential members.

Use the ::= operator to ENUMERATE the allowed actors in a configuration. Immutable actors should be identified by being underlined in the definition. All busses must have a seat and a pouch, which would be ENUMERATED as: $Bus ::= \{\underline{seat}, \underline{pouch_{Bus}}\}$.

A seat, on the other hand, is a configuration that may be empty or have an occupant. This will be modeled as: $seat ::= \{\{\}, Alice\}$

When a actor is declared to be a permanent part of a configuration, an implied policy is also created.

Use the ASSIGNment operator := to indicate a configuration's members. When a configuration contains actors, use a subscript to identify the parent and { } to demark the contents. In this case, the *Bus*'s current configuration is: *Alice* is on a seat in the *Bus* and there is no € in the *pouch* (which may violate a policy that will be defined later): $Bus := \{\{Alice\}_{\underline{seat}}, \{\}_{\underline{pouch_{Bus}}}\}$.

Table 4.5: The Notation Used in this Thesis

| **Operator** | **Definition** | **Usage** |
|---|---|---|
| $\wedge$ $\vee$ | TEST: Logical AND & OR. | If this and that or zig and zag...: $(this \wedge that) \vee (zig \wedge zag)$ |
| $\in$ $\ni$ | TEST: Is element $a$ a member of set $S$? Used to determine if an actor is in a configuration. | Alice has bus fare: $€ \in Alice$ |
| $\sqsubseteq$ $\sqsupseteq$ | TEST: Is the left term a subset (or superset) of the right term (inclusive of the root node). This is used to see if the left configuration is in the right's configuration. | Is the seat empty? $emptySeat \sqsubseteq seat$ |
| $\sqsubset$ $\sqsupset$ | TEST: Same as above except exclusive of the root node. | Does the bus have an empty seat? $emptySeat \sqsubset Bus$ |
| ::= | ENUMERATE a configuration. i.e. This operator identifies all possible members. | A seat may be empty or occupied by a person: $seat ::= \{\{\}, \boldsymbol{person}\}$ |
| := | ASSIGN actors or types to the left term. | To put Alice in the seat: $seat := \{Alice\}$ |
| == | TEST: Are the left and right sides equivalent? | Is the seat empty? $emptySeat == seat$ |
| $\sigma(a, b)$ | Count the number of instances of $a$ in configuration $b$. | How much money is in the pouch? $\sigma(€, pouch)$ |
| $\ltimes$ $\rtimes$ | ENUMERATE the allowed members of a type. | Alice and Bob are people. $\boldsymbol{person} \rtimes \{Alice, Bob\}$ |
| $\oplus$ | Attach a configuration. See Definition 6[34]. | To place $X$ underneath $Z$: $Z \oplus X$ |
| $\ominus$ | Remove a configuration. | To remove $X$ from $Z$: $Z \ominus X$ |
| $[Y \ominus X, Z \oplus X]$ | Move configuration X from Y to Z. See Figure 4.2 | Alice gives € to the bus: $[pouch_{Alice} \ominus €, pouch_{Bus} \oplus €]$ |
| © | Copy a configuration. | Make a copy of $X$ and call it $X_{copy}$: $X_{copy}$ © $X$ |
| $p : Y \xrightarrow{X} Z$ | A principle causes X to move from Y to Z. | Alice pays the bus fare: $Alice : pouch_{Alice} \xrightarrow{€} pouch_{Bus}$ |
| $\Rrightarrow$ | Guard. The expression on the left must be true to allow the move on the right to proceed. | Al can only take an empty seat: $(seat == \{\}) \Rrightarrow (\boldsymbol{Al}) : seat \xrightarrow{\boldsymbol{Al}} \{Al\}_{seat}$ |
| $\forall$ | For all. An assertion that everything to the right gets applied to the left. | Hash all of the memory in $ueficode$: $\forall m \in ueficode, hash(m)$ |
| $\neg$ | Negation. In plain language "not something". | If the seat is not empty: $seat == \neg emptySeat$ |

Table 4.6: Terminology Update

| **Term used in [34]** | **Term used in this Thesis** |
|---|---|
| configuration | set |
| box configuration | configuration |
| multiplicity or $mult(actor, configuration)$ | $\sigma(actor, configuration)$ |
| move | action |

Configuration ASSIGNment is an *in a* relationship. Alice is in a Bus, but Alice is not a Bus.

### 4.3.2 Type Notation

It takes two steps to define a type. The first step defines its use within a configuration. The second step defines the allowed members of the type.

To ENUMERATE the *use* of a type, we use the same notation we used to assign an actor, substituting a type in its place. For example, any person can take a seat on a bus. An improved definition of a *seat* is: $seat ::= \{\{\}, \boldsymbol{person}\}$.

To ENUMERATE the *allowed members* of a type, we need to map actors into it. The notation we will use is: $\boldsymbol{type} \rtimes \{actor_1, actor_2, ..., actor_n\}$. For example, actors *Alice* and *Bob* are of type $\boldsymbol{person}$, this would be expressed as: $\boldsymbol{person} \rtimes \{Alice, Bob\}$[8]. Note that the open end of the operator points toward the name of the type and the closed end points toward the set of allowed actors.

Type membership is a directional *is a* relationship. *Alice* is a $\boldsymbol{Person}$, but not all people are Alice. Use configurations to define an *in a* relationship.

Finally, types can be used in the abstract by constraining what configuration can be a member of another configuration.

## 4.4 Axioms

This thesis extends the work in [40, 34]. The axioms (established mathematical statements) are in Appendix B. Table B.1 summarizes the axioms and applies the terminology set out in Table 4.6.

Many axioms depend on a root node that ties everything together. From a modeling perspective, the root node may not add a lot of value, but from a proof perspective, many axioms depend on it.

Unfortunately, the source material does not provide all of the axioms needed in this thesis. Therefore, we must define one more, a copy axiom.

**Axiom 1.** We define a copy of configuration $X$ as $X_{copy}$, which are two separate actors such that $X == X_{copy}$ and $X_{copy} == X$. We create a copy with the © operator. Similar to assignment, the target is on the left and the operand on the right is read-only: $X_{copy}$ © $X$.

[8]This is abuse of notation for the sake of simplicity. In the source material, this would be: $\boldsymbol{person} \rtimes t \mid t \in \{Alice, Bob\}$. Where $t$ is a temporary placeholder.

## 4.5 Methodology

Actor Network Theory[32] resists formal methods, however we respectfully submit this methodology in the service of science. Actor Network Theory has some powerful concepts that, when combined with logic and set theory may give scientists tools to analyze the area between what a protocol *should* do and what it *can* do.

Now that we have identified the terminology and notation, we can describe the process used to model a protocol. An Actor Network Model is composed of two major parts: 1) Clearly defining the essential elements of the protocol and assigning them to various roles in the Actor Network: Principals, Actors, Configurations and the like. 2) A comparative analysis of what the network *can* do vs. what it *should* do. Oftentimes, the network is represented as a drawing to facilitate human comprehension; however, the important details of the network are a set of mathematical statements.

This Methodology may resemble the syntax of a programming language, however it is not our intent to define a strict grammar, which would be a disservice to Actor Network Theory. It does, however, borrow some principles from Software Engineering. For example, the notion of types is analogous to classes. The methodology attempts to strike a balance between simplicity and expressiveness – attempting to highlight important elements and abstract the rest. In this section, we will borrow from the Waterfall software development methodology in that it's unlikely that an Actor Network model will be fully specified in one pass. Modelers are encouraged to return to various "pages" of the model as it's continuously refined. As the model evolves, care should be taken to ensure that all documented rules continue to hold true.

Actor Network modeling defies a linear step-by-step process. Modelers can initially IDENTIFY a starting set of actors, but it's unlikely they can IDENTIFY all of them. In an effort to break the model into functional areas, the concept of Pages in introduced. Each page has dependencies on other pages. Pages can be developed in any order and may reference items in any other page[9]. A model is a collection of the following pages:

- Protocol Page
- Principal Page
- Actor Page
- Type Page
- Configuration Page
- Action Page
- Policy Page
- Analysis Page

### 4.5.1 The Protocol Page

This page introduces the protocol and describes what it should and shouldn't do.

- Give the protocol a name.
- In plain language, document the benefit of the protocol to the principals.
- In mathematical terms, DESIGNATE the goals of the protocol. This is definitive statement of what it *should* do.
- DESIGNATE any non-goals or things it expressly shouldn't do.

[9]Which is why the pages are named and not numbered.

### 4.5.2 The Principal Page

Promote selected actors to principals. All principles are actors, but if they are a principle, they should be documented here and not in the Actor Page. Principals are labeled with a leading capital letter i.e *Alice* instead of *alice* or ALL CAPS.

- IDENTIFY the principles. It should have *both* a label and brief description of the principle. If it's appropriate, identify which actors the principal controls.
- ENUMERATE the contents of the principle. All permanent (non-mutable) configurations should be identified with an underline.
- ENUMERATE any non-mutable or permanent configurations of the principals.
- If appropriate, add a drawing to help the reader understand the principals of the protocol.

### 4.5.3 The Actor Page

All other actors should be IDENTIFIED on this page.

- IDENTIFY every actor with a label and definition. If the label changes, be sure to update all of its references so that the model remains consistent and clear throughout.
- ENUMERATE the configuration of each actor. Use underlines to identify non-mutable or permanent members. Between the Principles Page and the Actor page, all non-leaf actors should have a configuration enumerated. True leaf-node actors don't need a configuration – they have no structure and the label is all that is needed.

At the time of this writing, I am undecided on a nomenclature for cardinality. For example, the enumeration could be "Choose none-or-one" or "Choose one" or "Choose n" or "Choose up to n". These are all important logical distinctions and I'd like to have a notation for them, but I'm not sure what way to go.

### 4.5.4 The Type Page

Types are an abstraction layer for Actor Networks. They allow actors to be substituted for one another. They also allow the Actor Network to be more expressive; the math will bear a closer resemblance to the plain language. Types are powerful because they can greatly expand what a network *can* do and uncover properties of the model that may not have originally been apparent.

Types have two enumerations. The first ENUMERATION is the *is a* relationship. It identifies what actors *are* a given type. The second ENUMERATION is the *has a* relationship. It identifies what actors the type *has*.

- IDENTIFY the types with a label and definition. Types don't have to be represented as actors.
- ENUMERATE type membership. All members should be an actor or another type. Type membership should not be empty. Members conform to the *is a* rule. In other words, a $\boldsymbol{place}$ could be a

$\{source, dest, town\}$ because they are all places. However, a $\boldsymbol{person}$ follows the *in a* rule. A person is not a place, but a person can be in a place.

- ENUMERATE type contents. A type may be required or allowed to *have* certain actors. A pouch may be empty or have some money in it: $\boldsymbol{pouch}$ ::= $\{\{\}, €\}$. Like the ENUMERATION for actors, use the underline to identify required contents.

### 4.5.5 The Configuration Page

This page focuses on the initial configuration of the model. From there, the actions and policies should elucidate any other states worthy of analysis. For clarity, a drawing may be helpful for this page, however, the configurations should also be expressed mathematically.

- For every actor that has a configuration, ASSIGN an initial set of actors to it.
- ASSIGN any constant actors.

Note: In [40], circles were used to diagram membership in a configuration. In [34], membership is diagramed as a hierarchy. This thesis will diagram configurations as hierarchies as that best conforms with the underlying mathematics.

### 4.5.6 The Action Page

All of the things that *can* be done are identified here. Identifying the actions helps modelers think about the allowed set of actions and define them using the most appropriate actors. The mapping from what *can* be done to what *should* be done happens through actions.

- IDENTIFY all actions giving them a label and a description.
- DESIGNATE the action, identifying the source actor, the destination actor, the actor that is moving and the principal that can initiate the action.

### 4.5.7 The Policy Page

This is where actors, TESTS and actions come together to form policies. DESIGNATE allowed transactions and any policies that must hold true either before or after the transaction.

- Document the intent of the policy in common language.
- DESIGNATE the policy, identifying the conditions that must be true for the action to be permitted.

### 4.5.8 The Analysis Page

Once the model has been established, the next step is to mathematically examine the benefit of the model. This page contains the analysis of the protocol and compares the configurations it can get into vice what it should get into.

Technically, the analysis is not part of the model, rather it's an output of the model, however the entire context of the analysis is the model and the model exists to be analyzed, so the analysis becomes part of the whole.

- Analyze the performance of the model against the goals of the protocol. Does it do what it *should* do?
- Analyze the model against what it *can* do.

### 4.5.9 Modeling Notes

The following are some guidelines for modelers:

- Clearly document the purpose or benefit of the protocol being modeled. A byproduct of considering the benefit is that it usually identifies the principals and actions between them.
- Scope the inside/outside boundaries of the protocol to identify the protocol-level inputs and outputs of the model.
- Principals should be well understood with clearly defined boundaries.
- Some principals may need an identity. In Actor Network Theory, actors consist of a network of other actors. A modeler must identify the set of actors that is distinct from the other actors in the network.
- Principals must have control of their identity. This may be done by maintaining privacy and control of a set of secrets. It may also be done by the possession of a copy-resistant token, but if it is, the modeler should be aware that the 'token' can be given away.
- Within a given model, principals should not overlap with other principals. Actors in Actor Network Theory are always composites of other actors. The trick is to identify the principals and nodes (actors) at an appropriate granularity to support the model. Maintaining a focus on the benefit of the protocol being modeled helps to identify the appropriate granularity of actor. Each actor should have a direct role in carrying the 'benefit' of the protocol at some point. If an actor does not have a role in the benefit, it may be modeled too finely. If the modeler is having trouble proving the benefit in PDL, then the actors may be too coarse.
- Remove any actors, types, principles that do not add value to the model.

## 4.6 An Example of an Actor Network

Putting it all together, we present an Actor Network model of a city bus. Our principals are *Alice* and a *Bus*. A protocol is established where, upon boarding the bus, riders pay a bus fare of €1 and are then permitted to ride to a destination.

Readers are encouraged to consider how the model could be extended to have more seats on the bus or to allow Alice to pay for both herself and Bob when they board.

### 4.6.1 Example Protocol Page

Name: Alice rides the bus

Benefit: Alice can economically cross town and the Bus will earn money for transporting passengers.

---

The goal of this protocol is: $(Alice \sqsubset destination) \wedge (paidFare \sqsubset Bus)$

### 4.6.2 Example Principal Page

| **Principal** | **Description** |
|---|---|
| *Alice* | Alice wants to cross town, going from source to dest. She has € in a pouch for bus fare. |
| *Bus* | The Bus travels to various places around town. It has a single seat and a pouch for bus fare. |

---

$Alice ::= \{\underline{pouch_{Alice}}\}$

$Bus ::= \{\underline{pouch_{Bus}}, \underline{seat}\}$

### 4.6.3 Example Actor Page

| **Actor** | **Description** |
|---|---|
| € | Bus fare. An important characteristic of € is that there is only one 'copy' of it. It can be given from one actor to another, but it can not be duplicated and it can't be in more than one place at once. |
| $town$ | The root node for the model. |
| $source$ | A place where Alice starts her trip. |
| $destination$ | A place where Alice wants to go. |
| $pouch_{Alice}$ | Alice's pocketbook. |
| $pouch_{Bus}$ | The Bus's cashbox. |
| $seat$ | A seat on the bus. |
| $emptySeat$ | An empty seat on the bus. |
| $occupiedSeat$ | A seat with a person in it. |
| $paidFare$ | The bus fare has been paid. |

---

$town ::= \{\underline{source}, \underline{destination}\}$

$seat ::= \{emptySeat, occupiedSeat\}$

$emptySeat ::= \{\}$

$occupiedSeat ::= \{\underline{\boldsymbol{person}}\}$

$paidFare ::= \{€\}_{pouch_{Bus}}$

**Notes**

- € is one of the true leaf-nodes in this model.
- *source* and *destination* are ***places***, which are declared on Example Type Page.

### 4.6.4 Example Type Page

| Type | Description |
|---|---|
| $\boldsymbol{place}$ | A nice place. |
| $\boldsymbol{person}$ | A nice person. A person may be in a place. |
| $\boldsymbol{pouch}$ | A secure place to keep money. |

---

$\boldsymbol{place} \rtimes \{source, destination, town\}$

$\boldsymbol{person} \rtimes \{Alice\}$

$\boldsymbol{pouch} \rtimes \{pouch_{Alice}, pouch_{Bus}\}$

---

$\boldsymbol{place} ::= \{\{\}, \boldsymbol{person}, Bus\}$

$\boldsymbol{pouch} ::= \{\{\}, €\}$

### 4.6.5 Example Configuration Page

The initial configuration of this protocol is:

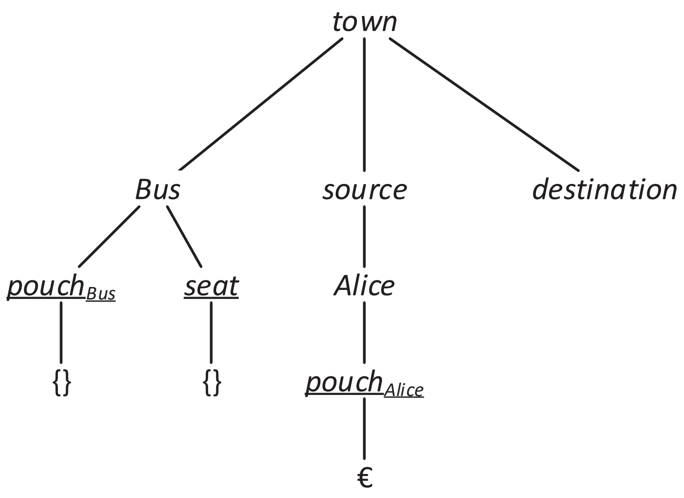


Figure 4.3: The Starting Configuration for this Protocol

In mathematical terms, the initial configuration is:
$\{\{\{\}_{\underline{pouch_{Bus}}}, \{\}_{\underline{seat}}\}_{Bus}, \{\{\{€\}_{\underline{pouch_{Alice}}}\}_{Alice}\}_{source}, destination\}_{town}$

### 4.6.6 Example Action Page

| **Type** | **Description** |
|---|---|
| $[boardBus]$ | Take a seat on the bus. $emptySeat \xrightarrow{\boldsymbol{person}} occupiedSeat$ |
| $[leaveBus]$ | Disembark from the bus. $occupiedSeat \xrightarrow{\boldsymbol{person}} emptySeat$ |
| $[payBusFare]$ | Pay the bus fare. $Alice : pouch_{Alice} \xrightarrow{€} pouch_{Bus}$ |
| $[drive]$ | The bus drives from place to place. $Bus : \boldsymbol{place_1} \xrightarrow{Bus} \boldsymbol{place_2}$ |

**Notes**

- No refunds. There is no action where the Bus pays Alice.

### 4.6.7 Example Policy Page

A person can only occupy an empty seat. Reading this expression; if the seat is empty, then a person who wishes to board a bus can move into an empty seat making it an occupied seat.

$(seat == emptySeat) \Rrightarrow (\boldsymbol{person}, Bus) : emptySeat \xrightarrow{\boldsymbol{person}} occupiedSeat$

...which can also be written as:

$(seat == emptySeat) \Rrightarrow (\boldsymbol{person}, Bus) : [boardBus]$

---

A person must be on the bus, to get off the bus. Some of these are obvious, but the math needs them.

$(seat == occupiedSeat) \Rrightarrow (\boldsymbol{person}, Bus) : occupiedSeat \xrightarrow{\boldsymbol{person}} emptySeat$

---

When the bus carrying Alice arrives at the destination, Alice will exit the bus.

$((Alice \sqsubset Bus) \wedge (Bus \sqsubset destination)) \Rrightarrow (\boldsymbol{person}, Bus) : [leaveBus]$

---

To board a bus, the following must hold:

- Both Alice and the Bus must be at the same place.
- There must be an empty seat on the bus.
- Alice must pay the bus fare.

Let $\ell \in \boldsymbol{place}$ Let $p \in \boldsymbol{person}$

$(Bus \sqsubset \ell) \wedge (p \sqsubset \ell) \wedge (emptySeat \sqsubset Bus) \wedge (paidFare \sqsubset Bus) \Rrightarrow (p, Bus) : [boardBus]$

### 4.6.8 Example Analysis Page

**Analysis of what the network *should* do**

- The only way that Alice can get to the destination is to pay the bus fare.
- There is no policy that requires the bus take Alice to the destination once she has paid the bus fare.

**Analysis of what the network *can* do**

- There is no policy that says the bus will go to the source to pickup Alice.

# CHAPTER 5
# MODELING THE SECURE BOOT PROCESS

boot — noun
A sturdy item of footware.

*New Oxford American Dictionary*

## 5.1 Introduction

### 5.1.1 Background

Like a lot of technical jargon, the word 'boot' has taken on a life of its own. Sometimes it's used as a verb as in "Please boot the computer" at other times, it's used as a noun such as "The BIOS contains the boot code". A computer follows a 'boot' process to put itself into a *stable*, known state. Colloquially it's the process of starting a computer.

The existence and widespread distribution of bootkits, rootkits and hardware viruses [28, 14, 52, 20] indicate increasing attempts to penetrate computer systems at lower levels in order to escape traditional observations. Even overwriting non-volatile random-access memory (nvRam) can trigger a Denial-of-Service attack [53]. This type of malware takes control before anti-virus software can detect it. Recent improvements to firmware standards [49, 47, 42] and the introduction of trusted boot processes [26, 35, 56, 45, 13, 51, 55] show the commercial need and interest to address and prevent such low level attacks.

[54] proposes "presence attestation" to improve the trust of Dynamic Root of Trust for Measurement (DRTM) based on sight, location or scenes all of which lends itself to Actor Network Modeling.

Trusted boot processes incorporate many layers and phases, each requiring the ability to be updated. Early implementations of trusted boot processes contained vulnerabilities demonstrating that the development of commercial trusted boot solutions is not straightforward and may be an error-prone task. Yet, describing the protocols and features of a specific trusted boot process in a rigorous framework that would allow for formal analysis is non-trivial and requires a methodology to not only model hardware and software components, but also people, identities and locations. Conventional models to describe computer hardware, software and network protocols typically fail to describe distinguishing physical features such as sensors, actuators and locations.

This section contains four Actor Network models that work together to bring a system from a cold-start to a fully operational system. The first model ensures the system is only executing $uefiCode$ – or the software we are about to 'measure' or 'analyze' and decide if it's trusted. The second model analyzes $uefiCode$ and decides if it's to be trusted or not. These two models look 'backward' into the same software that's running to determine if it's trustworthy or not. The third model is a forward looking model that evaluates software and determines its trustworthiness before the system is allowed to execute any of its code. The last model ensures that the firmware involved in this process must be updatable by authorized system administrators.

We use a cryptographic technique called "Digital Signatures" to determine if the software can be trusted. We do this by ensuring that the authors have used cryptographic controls to verify that the trusted software on the computer is exactly the same software that shipped from their factory. This guarantees that the source of the software is both known and trusted by the computer. The models make no guarantees of the quality of the software of the robustness of their implementation of these protocols.

Computers are very complex machines. When we model a particular protocol, it would be impractical to model the entire computer – the model would be too complex. Therefore, we must group certain details together into actors and ignore other details, which, although crucial to the computer, are not salient to the protocol. A grouping of details is abstracted into an actor. These actors interact with other actors to express a protocol which we model and analyze.

All four of these protocol models consist of different actor-networks that change depending on the model's needs. Each protocol creates different objects, measures different actors and protects actions differently. Consequently, our actors interact differently for each model.

A good example is the difference between how memory is modeled in *Secure UEFI Boot* and *Secure UEFI Validation*. In Secure UEFI Boot, the model is examining the addresses of memory in $ueficode$. One of the main goals of this protocol is to ensure that the CPU's Program Counter (pc)[1] always contains an address that is a member of $ueficode$. In the model, the actor $ueficode$ should be considered a set of addresses. In this protocol, if the *pc* ever contains an address that is not a member of $ueficode$, then the protocol fails and $ueficode$ should not be trusted. On the other hand, Secure UEFI Validation models the contents of memory. This protocol considers $ueficode$ a set of instructions and data.

The Secure Boot, Trusted Boot and Trusted Computing processes have several stages. Fortunately, many of the stages utilize a common pattern of measurement and execution. The very first boot processes use Secure UEFI Boot and Secure UEFI Validation to start the system and establish a root of trust. This is called "Secure Boot". Subsequently, nvRam configuration, UEFI drivers and Kernel loaders will use Secure Software Load to measure and extend the trust domain to that software.

The remainder of this chapter is divided into four Actor Network protocols:

- Secure UEFI Boot
- Secure UEFI Validation
- Secure Software Load
- Secure UEFI Update

Many of the models contain figures such as Figure 5.1. This diagram represents a "Hoare Triple"[22], which is a mathematical expression that shows the state BEFORE an action, labels the ACTION, and the

[1]Sometimes called an Instruction Pointer or IP.

state AFTER the action. In this diagram, there are two actions, [decrypt] and [hash]. The red color signifies that something is changed or is causing a change.

The network to the left of the → shows the state of the network BEFORE the action. The network to the right of the → shows the state of the network AFTER the action.

The straight lines between actors indicate membership (see Table 4.5). The curved lines with arrowheads indicate a flow of information from one actor to another.

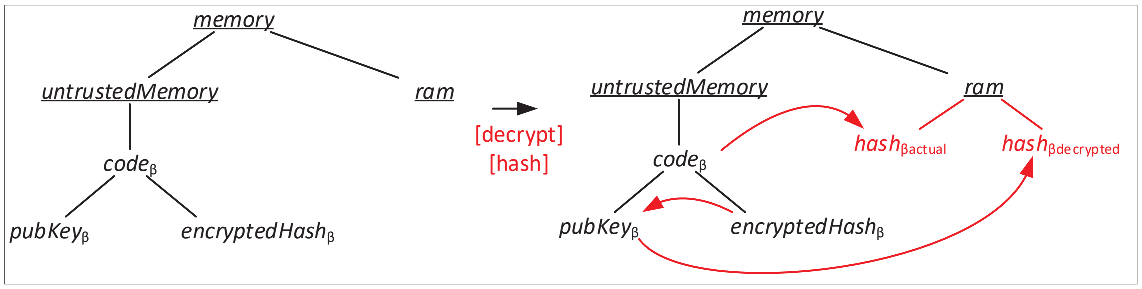


Figure 5.1: Sample Figure

### 5.1.2 Secure UEFI Boot Protocol

Our first model is also the simplest. All four of these models work together to bring the system from a cold-start to a fully operational system.

The purpose of Secure UEFI Boot is to model the simplest possible computer. All it has is a CPU and memory. The CPU, which is incredibly complex in the real world, is abstracted such that it only has one component that we are interest in: Its Program Counter. The program counter is a CPU register that points to a memory location of the current instruction.

Before we talk about memory, we need to understand a few basic CPU instructions: LOAD and STORE. The LOAD instruction takes the contents of memory at a specified address and LOADS it into a register in the CPU. The STORE instruction, on the other hand, takes the contents of memory and stores it in memory at a specified address, overwriting its original contents.

In this model, the memory is divided into three actors. The first is *rom*, or Read Only Memory. This is a type of memory that is indelibly programed in the factory and can not be changed without physically altering the computer. In the early days of computers, the first instructions the computer ran were often written in *rom*. We have *rom* in our model to explicitly say that we do not use it. One requirement of these protocols is that all of the software can be updated. Therefore, we do not use *rom*. CPUs can use LOAD with *rom* memory, but a STORE instruction will always fail.

What many people think of as *rom* is actually the second type of memory we model called *nvRam*. The ‘nv’ stands for Non-Volatile, in other words, the contents of this memory are retained even when it does not have power. The ‘Ram’ stands for Random Access Memory. Unqualified RAM is usually associated with a type of memory that *is* volatile and can not reliably save data without power. A CPU LOAD instruction works with nvRam. A STORE instruction will always fail with nvRam, but there is another way to write to it. ‘nvRam’ is updated with special I/O operations that write in blocks similar to the sectors of a hard drive. *ram*, our third type of memory, can use both the LOAD and STORE instruction.

There is another way to update memory, which is not exprssed any of our models. It’s called DMA or Direct Memory Access. DMA allows devices like hard disk controllers to write to certain blocks of *ram* – bypassing the CPU’s STORE instructions. DMA allows computers to be very efficient. An analyst could think of DMA, however, as a means to attack the system. The analyst should carefully observe when data is stored in memory without cryptographic protections or memory management controls.

Underneath nvRam, we have divided memory into two logical groupings: trustedMemory and untrustedMemory. When a computer is started, we consider the $uefiCode$, which is stored in *nvRam* to be $untrustedMemory$. The goal of this protocol is to move the $uefiCode$ to $trustedMemory$. We do this using the following Actions:

- We inductively assert that at all times during UEFI Boot (and UEFI Validation), the *pc*, points to addresses in $uefiCode$.
- We use our second protocol $SecureUEFIValidation$ to assert that the code is trusted (or not). (see Equation 5.2)
- Finally, if it is trusted, we use our Actor Network operators from Table 4.5 to move $uefiCode$ to $trustedMemory$.

Our inductive proof is adapted from [22, 18] where we know what the *pc* is and we know what the *pc* was for the previous instruction. We do not know what the next instruction is. So long as the current and previous instruction are addresses in $uefiCode$, this relationship will roll backwards to the very first instruction, which, if it is also in $uefiCode$, then we can prove that all of the instructions executed up to this point are in $uefiCode$ (see Equation 5.1).

What follows is a model and proof that allows the $uefiCode$ to link to $trustedMemory$.

**Secure UEFI Boot Protocol Page**

Name: Secure UEFI Boot
Benefits:

A. Establish trust for the UEFI firmware that is running just after boot.

---

The goal of this protocol is: $uefiCode \in trustedMemory$

**Notes**

- This is the smallest first-step we can make. It works in conjunction with UEFI Verify. I separated these two protocols to simplify the first few production Actor Network models.
- In this model, references to *memory* refer to addresses. In the UEFI Verify model, references to *memory* refer to the data stored in *memory*.
- This protocol is distinct from Secure Software Load in that this (and UEFI Boot) are running the same code it's measuring. Secure Software Load will measure the code first and then run it.
- $uefiCode$ does not, as yet, read any persistent data such as nvRAM.

**Secure UEFI Boot Principal Page**

| **Principal** | **Description** |
|---|---|
| $CPU$ | The central processing unit of the computer. |

---

$CPU ::= \{\underline{pc}\}$

**Notes**

- We limit our model to a single CPU.

## Secure UEFI Boot Actor Page

| Actor | Description |
|---|---|
| *computer* | All of the hardware and software on a given computer. |
| *memory* | Memory that's available to the computer. |
| *pc* | The Instruction Pointer in the $CPU$. Sometimes called a Program Counter. |
| $addr_0, addr_1, ..., addr_n$ | All possible *memory* addresses for this *computer*. The subscript represents the memory address. $addr_0$ represents address 0x0000. |
| *nvRam* | Non-Volatile RAM. Normally, the UEFI / BIOS would be in ROM, but then it could not be updated. In this model, it's in *nvRam*, which requires special processes (outside of normal CPU Store Memory commands) to update. |
| $untrustedMemory$ | Configurations under this actor should not be trusted. |
| $trustedMemory$ | Configurations under this actor have been trusted by the system. |

| Leaf Node Actor | Description |
|---|---|
| *rom* | Traditional read-only memory. This is modeled without any members to indicate that no ROM exists in the computer. All firmware can be updated. |
| *ram* | Traditional read-write memory. It is not modeled, although it may be used as a scratchpad for uefiCode. |
| $uefiCode$ | The set of every memory address (and its contents) that make up the BIOS / UEFI firmware. It is this memory region the protocol is attempting to trust. |
| $cpuStartAddr$ | The starting address of the $CPU$. $cpuStartAddr \in uefiCode$ |

---

$computer ::= \{\underline{CPU}, \underline{memory}\}$

$CPU ::= \{\underline{pc}\}$

$memory ::= \{rom, \underline{nvRam}, ram\}$

$nvRam ::= \{untrustedMemory, trustedMemory\}$

$pc ::= \{addr_0, addr_1, ..., addr_n\}$

## Notes

- For $addr_n$, $n$ represents the highest memory address.
- $uefiCode$ is not writable. Any attempt to change the code, at any time, will invalidate the model.

## Secure UEFI Boot Type Page

| **Type** | **Description** |
|---|---|
| $\boldsymbol{memoryAddr}$ | An address of memory. |

---

$\boldsymbol{memoryAddr} \rtimes \{addr_0, addr_1, ..., addr_n, \{uefiCode\}, cpuStartAddr\}$

---

None

### Notes

- In this model, all $\boldsymbol{memoryAddr}$s are leaf nodes.

## Secure UEFI Boot Configuration Page

The initial configuration of this protocol is:

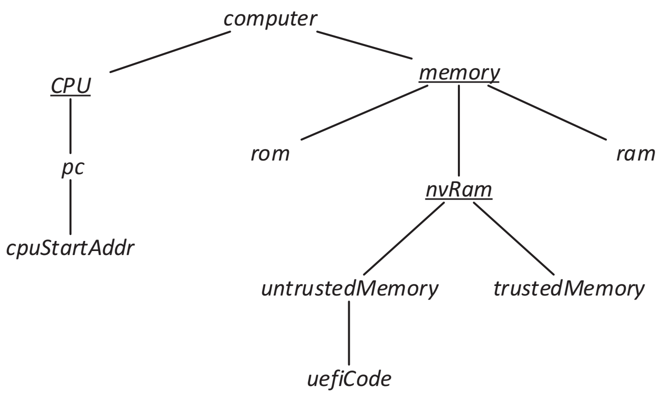


Figure 5.2: Initial Configuration at UEFI Boot

In mathematical terms, the initial configuration is:

$computer := \{\underline{CPU}, \underline{memory}\}$

$CPU := \{pc\}$

$pc := \{cpuStartAddr\}, cpuStartAddr \in uefiCode$

$memory := \{rom, \underline{nvRam}, ram\}$

$memory := \{untrustedMemory, trustedMemory\}$

$untrustedMemory := \{uefiCode\}$

**Secure UEFI Boot Action Page**

| Type | Description |
|---|---|
| $[execute]$ | Execute an instruction. <br> $CPU : \{pc == addr_{nPrev}\} \xrightarrow{CPU} \{pc == addr_n\}$ |
| $[uefiValidate]$ | Modeled in section 5.1.3. |
| $[trust]$ | Trust the $uefiCode$. <br> $CPU : \{uefiCode \in untrustedMemory\} \xrightarrow{CPU}$ <br> (cont.) $\{untrustedMemory \ominus uefiCode, trustedMemory \oplus uefiCode\}$ |

**Notes**

- $pc$ must be $\in uefiCode$. This gets asserted in the next section.
- The model takes some liberties with respect to $x$ overrunning $n$ or underrunning 0. For the purposes of the model, we will assume that $pc$ is $MOD(n + 1)$ and if the new address is still $\in uefiCode$ then the code will continue to [execute].
- In Figure 5.3 demonstrates the linear execution of a 2-byte instruction that does not branch.

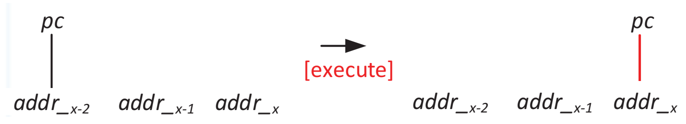


Figure 5.3: Executing $uefiCode$

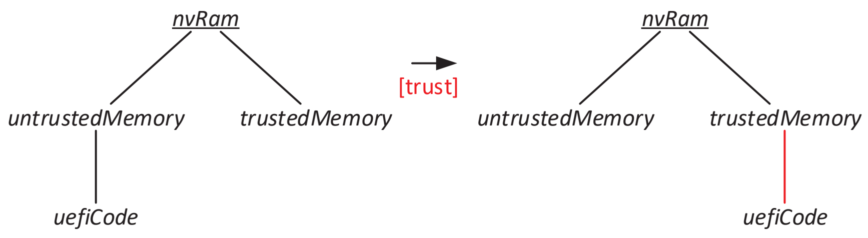


Figure 5.4: Trusting $uefiCode$

**Secure UEFI Boot Policy Page**

$pc$ is always an element of $uefiCode$. Each execution changes the $pc$ from $addr_{nPrev}$ to $addr_n$. This execute transition is enabled if both $addr_{nPrev}$ and $addr_n$ are in $uefiCode$.

$$\begin{aligned} cpuStartAddr \in uefiCode \land pc \in uefiCode :& \\ (addr_{nPrev} \in uefiCode \land addr_n \in uefiCode) \Rrightarrow (CPU) : \{pc == addr_{nPrev}\} \xrightarrow{CPU} \{pc == addr_n\}& \end{aligned} \tag{5.1}$$

Which is the same as:
$(addr_n \in uefiCode) \Rrightarrow (CPU) : [execute]$

---

If $uefiCode$ is trusted by $[uefiValidate]$ then trust it.

$$\begin{aligned} (trusted == [uefiValidate]) \Rrightarrow (CPU) : \{uefiCode \in untrustedMemory\} \xrightarrow{CPU}& \\ \{untrustedMemory \ominus uefiCode, trustedMemory \oplus uefiCode\}& \end{aligned} \tag{5.2}$$

---

$uefiCode$ should not be writable. If any $uefiCode$ changes, then it invalidates this model.

---

**Secure UEFI Boot Analysis Page**

**Analysis of what the network *should* do**

- $pc$ should never escape $uefiCode$.
- If $[uefiValidate]$ determines that $uefiCode$ $\underline{\text{at one point in time}}$ is the same as the code in the factory or from a secure update.
- AND if $uefiCode$ is indelible, then it should remain trusted for the running time of the $computer$.

$pc \in uefiCode$

$(trusted == [uefiValidate]) \Rrightarrow (CPU) : \{uefiCode \in untrustedMemory\} \xrightarrow{CPU} \{untrustedMemory \ominus uefiCode, trustedMemory \oplus uefiCode\}$

**Analysis of what the network *can* do**

- There are no protections around the *ram*, which will be used as the stack. This model makes no guarantees that *ram* values are not altered after they are written but before they are read, nor does this model protect against alteration in transit. For example, *ram* data is not digitally signed.

### 5.1.3 Secure UEFI Validation Protocol

The goal of the next two protocols is to decide if software is trustworthy or not. In this case, we are examining the same $uefiCode$ that the protocol is running. This is a ‘backwards’ looking test of trustworthiness.

In this model, the *computer* has three actors, a simple CPU, some memory and a new device called a TPM. The content of memory is $uefiCode$. Practically, $uefiCode$ may include BIOS or UEFI code, it may include Option ROM if there’s a possibility it may execute while this protocol is running. The $uefiCode$ does not need to be in a contiguous address space. It can be spread out anywhere in addressable memory or shadow memory. The important property of this actor is that any and all code that the $CPU$ may execute during this phase must be a member of $uefiCode$ or the protocol is invalid.

A Trusted Platform Module or TPM is a tamper-resistant device whose purpose is to help verify trust in the platform. A trusted system is one where, for each component, either hardware or software, the provenance can be guaranteed and the user of the system elects to trust the originator. A trusted system is no guarantee of correctness (liveness), safety or security. There is some debate as to the trustworthiness of TPMs and their role in Digital Rights Management, loss of anonymity and control of your system. This thesis simply tries to model the TPMs, UEFI and system architectures as they are used today.

The TPM contains two interrelated actors that are critical for establishing trust: nvIndexes and PCRs. nvIndexes (the way we use them) are small, secure blocks of memory that store a few bytes of information. In our case, we store a version number. The nvIndexes are protected by numerous TPM policies, but for this model, this Actor Network will enforce a simple read policy. When the nvIndex is created, it is assigned a set of rules, or policies in TPM-nomenclature, that govern what conditions must be true for the TPM to allow anyone to read the nvIndex. For our model, the nvIndex is protected by a policy that says that $pcr_0$ must be a certain value. That’s it... if $pcr_0$ is that value, then you can read nvIndex. If $pcr_0$ is anything else, then the TPM will not allow nvIndex to be read.

How do you get $pcr_0$ to that value? TPMs are protected by not allowing PCRs to be set directly. Equation 5.3 shows how PCR values get set through a function called *extend*(). PCRs use a cryptographic hash() to measure software state. A PCR’s value can not be ‘rolled back’. For a PCR to get to a certain value, a series of extend(value) functions must be executed in order. For the PCR to return to that value, after a reboot, the PCR must receive an identical set of extend(value) function calls in the same order. If the value represents a hash of the $uefiCode$, then as long as the current PCR value matches a trusted value stored in the TPM, for example, the value protecting an nvIndex, then the system has some information to determine the trustworthiness of the code.

If the $pcr_0$ value allows the nvIndex to release the version number AND if that version number is in the $uefiCode$ that was part of the hash that went into $pcr_0$, then the *computer* will assert that $uefiCode$ is trustworthy – it has not changed since it left the factory or was updated through the Secure UEFI Update Protocol. This is expressed in Equation 5.4.

$$extendPCR(parm_1) : pcr_{new} := hash(pcr_{old} \| parm_1) \tag{5.3}$$

**Secure UEFI Validation Protocol Page**

Name: Secure UEFI Validation
Benefits:

A. Validate that the UEFI Code is trusted by the TPM.

---

The goal of this protocol is: $(\boldsymbol{version_{nvIndex}} == \boldsymbol{version_{ueficode}})$

**Notes**

- This is the smallest first-step we can make.
- This protocol is distinct from Secure Software Load in that this (and UEFI Boot) are running the same code it's measuring. Secure Software Load will measure the code first and then run it.
- The root of trust is the TPM.
- This analysis assumes that the current UEFI code has been loaded and sealed into the TPM's PCRs either at the factory or through a secure update protocol.
- The code does not, as yet, use any data such as nvRAM.

**Secure UEFI Validation Principal Page**

| **Principal** | **Description** |
|---|---|
| $TPM$ | Trusted platform module. |
| $CPU$ | The central processing unit of the computer. |

---

$TPM ::= \{\underline{pcr_0}, nvIndex_0\}$
$CPU ::= \{\boldsymbol{hash}\}$

**Notes**

- We limit our model to a single CPU.
- TPMs must have PCRs, but they are not required to have populated nvIndexes.

**Secure UEFI Validation Actor Page**

| **Actor** | **Description** |
|---|---|
| *computer* | All of the hardware and software on a given computer. |
| *memory* | Memory that's available to the computer. |
| $uefiCode$ | The set of every memory address (and its contents) that make up the BIOS / UEFI firmware. It is this memory region the protocol is attempting to trust. |
| $pcr_0$ | Core root of trust measurement (CRTM). It contains the measurement of the BIOS / UEFI firmware in the form of a $\boldsymbol{hash}$ that has been 'extended' by other $\boldsymbol{hash}$s. |
| $nvIndex_0$ | Contains a $\boldsymbol{version}$ of the BIOS / UEFI firmware and a copy of $pcr_0$ at the time the BIOS / UEFI firmware was trusted. Access is protected by $pcr_0$. This value is set either in the factory or through a secure update protocol. |

| **Leaf Node Actor** | **Description** |
|---|---|
| $reset$ | The uninitialized state of an actor. |
| $hash_{uefiCode}$ | The hash of $uefiCode$. |
| $pcr_{0uefiCode}$ | The PCR-extended value of $hash_{uefiCode}$. |
| $pcr_{0VersionX}$ | A saved copy of $pcr_0$. When $pcr_0 == pcr_{0VersionX}$, then the $TPM$ will allow $nvIndex_0$ to be read. |
| $versionX_{nvIndex_0}$ | The UEFI version number stored in $nvIndex_0$. |
| $versionX_{uefiCode}$ | The UEFI version number stored in $uefiCode$. |
| $versionX_{fromTPM}$ | The data stored in $nvIndex_0$ that is copied to $memory$. |

---

$computer ::= \{\underline{TPM}, \underline{memory}, \underline{CPU}\}$

$memory ::= \{\underline{uefiCode}, \boldsymbol{hash}, \boldsymbol{version}\}$

$uefiCode ::= \{\underline{versionX_{uefiCode}}\}$

$pcr_0 ::= \{reset, \boldsymbol{hash}\}$

$nvIndex_0 ::= \{reset, \boldsymbol{hash}\}$

**Notes**

- The $\boldsymbol{hash}$ stored in *memory* is in uninitialized RAM. The hash's state is written to RAM *first* and accessed later. No RAM is read before it is initialized. In other words, in this context, it's a scratchpad.
- This protocol assumes that $\boldsymbol{hash}$ and $\boldsymbol{version}$ are an accurate copy of their source and they are not modified between when they are written and later read. Conversely, there is some risk that another entity could update them as in memory.
- $uefiCode$ contains both executable code and standing data such as $versionX_{uefiCode}$, however only the latter is vital to this model.

**Secure UEFI Validation Type Page**

| Type | Description |
|---|---|
| $\boldsymbol{pcr}$ | Platform Control Register. |
| $\boldsymbol{nvIndex}$ | Non-volatile RAM in the TPM that is protected by policy. |
| $\boldsymbol{hash}$ | A cryptographically secure hash value := $hash(message)$ |
| $\boldsymbol{version}$ | The version number of something the protocol is attempting to trust. |

---

$\boldsymbol{pcr} \rtimes \{pcr_0\}$

$\boldsymbol{nvIndex} \rtimes \{nvIndex_0\}$

$\boldsymbol{hash} \rtimes \{hash_{ueflCode}, pcr0_{ueflCode}, pcr0_{VersionX}\}$

$\boldsymbol{version} \rtimes \{versionX_{nvIndex_0}, versionX_{ueflCode}, versionX_{fromTPM}\}$

---

$\boldsymbol{pcr} ::= \{reset, \boldsymbol{hash}\}$

$\boldsymbol{nvIndex} ::= \{\underline{\{\boldsymbol{version}, \boldsymbol{hash}\}}\}$

**Notes**

- The hash function is a cryptographic one-way function in that its security property is it that it is prohibitively hard (due to size and computational complexity) for an attacker to deliberately generate a collision.
- $\boldsymbol{nvIndex}$'s do not have a $reset$ state because this model assumes the $\boldsymbol{nvIndex}$ will be set in the factory and never reset after that.
- In this model, all $\boldsymbol{hash}$s and $\boldsymbol{version}$s are leaf nodes.
- *Technically* $\boldsymbol{pcr}$ and $\boldsymbol{nvIndex}$ do not need to be types as there is only one actor that instantiates them. However, real TPMs have several nvIndexes and PCR registers, so this form is more general.

**Secure UEFI Validation Configuration Page**

The initial configuration of this protocol is:

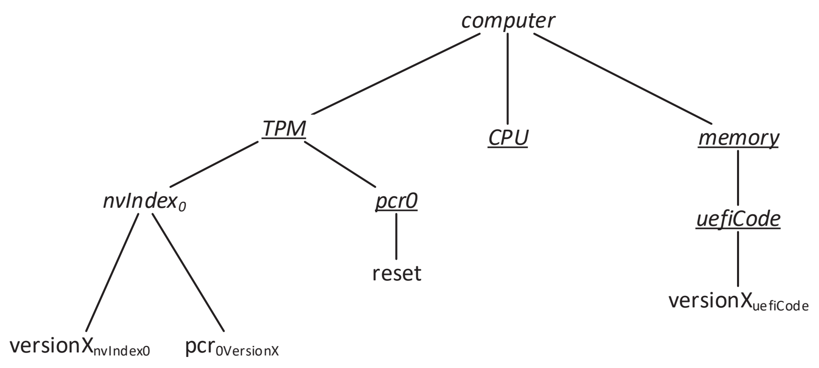


Figure 5.5: Initial Configuration Before UEFI Validation

In mathematical terms, the initial configuration is:

$computer := \{\underline{TPM}, \underline{CPU}, \underline{memory}\}$

$TPM := \{nvIndex_0, pcr_0\}$

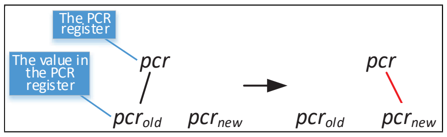
$nvIndex_0 := \{versionX_{nvIndex0}, \underline{pcr_{0VersionX}}\}$

$pcr_0 := \{reset\}$

$memory := \{\underline{uefiCode}\}$

$uefiCode := \{versionX_{uefiCode}\}$

**Secure UEFI Validation Action Page**

| Type | Description |
|---|---|
| $[hash]$ | A secure hash function. $CPU : ueficode \xrightarrow{CPU} hash_{ueficode} := hash(ueficode), memory \oplus hash_{ueficode}$ |
| $[extend]$ | A TPM primitive that accepts a hash as input and makes the following computation: $\boldsymbol{pcr}_{new} := hash(\boldsymbol{pcr}_{old} \| inputHash)$. $CPU, TPM : \boldsymbol{inputHash}, \boldsymbol{pcr}_{old} \xrightarrow{TPM} \boldsymbol{pcr}_{new} := hash(\boldsymbol{pcr}_{old} \| \boldsymbol{inputHash})$, $\boldsymbol{pcr} \ominus \boldsymbol{pcr}_{old}, \boldsymbol{pcr} \oplus \boldsymbol{pcr}_{new}$   The PCR register<br>The value in the PCR register<br>pcr<br>pcr_old<br>pcr_new<br>pcr<br>pcr_old<br>pcr_new |
| $[nvRead]$ | Read the value stored in nvIndex and put a copy in main memory. The ability to read this value is protected by a policy enforced by the $TPM$. $TPM : \boldsymbol{version}_{inNvIndex} \xrightarrow{TPM} \boldsymbol{version}_{copy} © \boldsymbol{version}_{inNvIndex}$, $memory \oplus \boldsymbol{version}_{copy}$ |

**Notes**

- PCR's can not be written to directly, they can only be extended. The extend function is a deterministic, one-way function that hashes it's old value appended with a new hash[47].
- We assume that the $hash()$ function is such that collisions are impossible to force, thus providing assurance that when two hash values are the same, then the two inputs are also the same.

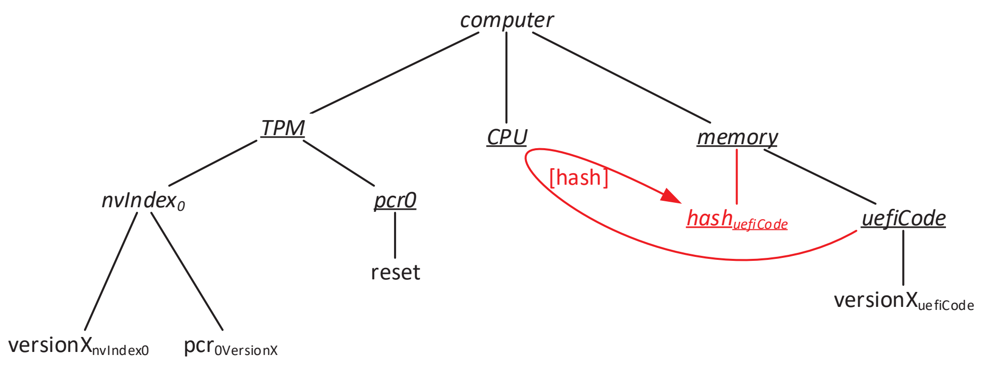


Figure 5.6: Hashing $ueficode$

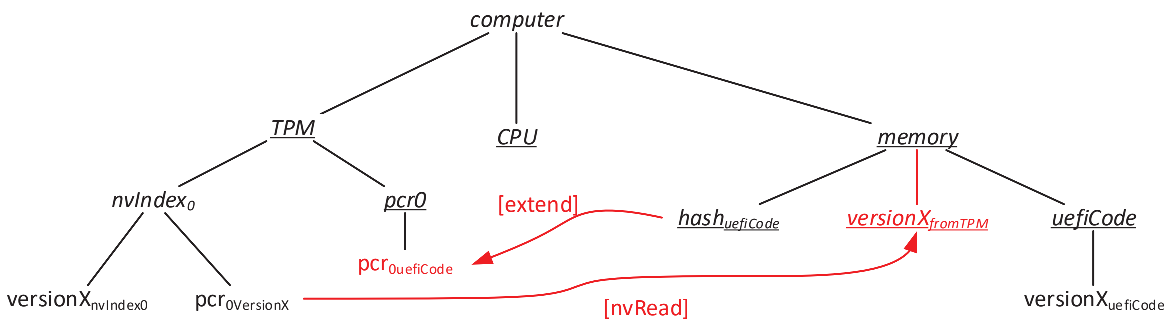


Figure 5.7: Extending $pcr_0$ and reading $nvIndex_0$

**Secure UEFI Validation Policy Page**

The hash function must hash all of uefiCode:

$hash_{uefiCode} := hash(uefiCode)$

(not modeled) The hash function must process uefiCode in the same order as the factory and/or future validations.

---

The value stored in $nvIndex_0$ can only be read if $prc_0$ matches the $\boldsymbol{pcr}$ value sealed in $nvIndex_0$.

$(pcr_{0VersionX} == pcr_{0uefiCode}) \Rrightarrow (TPM) : \boldsymbol{version}_{inNvIndex} \xrightarrow{TPM} \boldsymbol{version}_{copy} \copyright \boldsymbol{version}_{inNvIndex}$, $memory \oplus \boldsymbol{version}_{copy}$ Which is the same as: $(pcr_{0VersionX} == pcr_{0uefiCode}) \Rrightarrow (TPM) : [nvRead]$

---

If the UEFI version read from the TPM matches the UEFI version contained in $uefiCode$ then the $CPU$ will determine that $uefiCode$ is trusted.

(5.4)

$(versionX_{uefiCode} == versionX_{fromTPM}) \Rrightarrow (CPU) : uefiCode$ is trusted.

---

**Secure UEFI Validation Analysis Page**

**Analysis of what the network *should* do**

- The system can determine if $uefiCode$ at one point in time is the same as the code in the factory or from a secure update.

$hash_{uefiCode} := hash(uefiCode)$

$(CPU, TPM) : \{hash(uefiCode), pcr_{0reset}\} \xrightarrow{TPM} \{pcr_{0uefiCode} := hash(pcr_{0reset} \,\|\, hash_{uefiCode})$, $pcr_0 \ominus pcr_{0reset}, pcr_0 \oplus pcr_{0uefiCode}\}$

$(pcr_{0VersionX} == pcr_{0uefiCode}) \Rrightarrow (TPM) : \{versionX_{nvIndex0}\} \xrightarrow{TPM} \{versionX_{fromTPM} \copyright versionX_{nvIndex0}$, $memory \oplus version_{fromTPM}\}$

$(versionX_{uefiCode} == versionX_{fromTPM}) \Rrightarrow (CPU) : uefiCode$ is trusted.

**Analysis of what the network *can* do**

- There are no protections around the two intermediate values: $versionX_{fromTPM}$ and $hash_{uefiCode}$. This model makes no guarantees that these values are not altered after they are written but before they are read, nor does this model protect against alteration in transit. For example, this data is not digitally signed by the TPM.
- There are no protections around the $uefiCode$ in $memory$ being altered after the initial $hash()$ function.

### 5.1.4 Secure Software Load Protocol

This protocol starts with programs and data that's already in *trustedMemory*. The goal of this protocol is to establish the trustworthiness of some new software. This software is broadly defined and may originate from a *FACTORY* which is modeled, but may originate within *trustedMemory* and is securely stored for later retrieval. The software may be drivers, loaders, programs, data, lists of other certificates or just about anything you can map into a computer's memory.

Like the other models, this protocol is only allowed to run software in *trustedMemory*.

This model has an Actor Network that's rooted with the *world*. The *world* contains a *FACTORY* and a *computer*. The *FACTORY* produces software, digitally signs it and ships it into the world. The *computer* loads the software from the *world* and verifies the digital signature to determine if it's trustworthy.

The protocol consists of three phases: Pre-loading of trusted keys (not modeled), activity in the *FACTORY* and what happens in the *computer*. This protocol uses asymmetric cryptography (public & private encryption keys) for its digital signatures. Trust between the *FACTORY* and the *computer* must be established before the protocol begins and is not modeled. The trust starts with a private key that's securely kept at the *FACTORY* and protected from unauthorized use. If the key is compromised, then another protocol, again not modeled, must be employed to revoke or "untrust" compromised keys. A set of public keys is pre-loaded on the *computer*. The specific method of storing the keychain is not modeled, but it is assumed to be trusted and secure[2]. If a piece of software has been signed by a private key and it's corresponding public key is one of the keys in this set (inclusive of superior keys in a hierarchy) then it is deemed to be trustworthy by the *computer*.

The second phase happens at the *FACTORY*. This is the first part of the model. The *FACTORY* has some new, unsigned software. In order to digitally sign the software, the software is digitally hashed[3] and then the hash is encrypted using the *FACTORY*'s private key. The original software is packaged with the public key(s) and the encrypted hash and is now a digitally signed piece of software that ships from the factory out into the world.

The last phase happens in the *computer* and is diagrammed in Figure 5.9. The software is loaded from the *world* into *untrustedMemory*. The *computer* then extracts the public key from the software and determines if it is in the *computer*'s list of trusted public keys. If it is not, then the protocol ends and the software remains in *untrustedMemory*.

If the public key is trusted, then the protocol will extract the encrypted hash from the program, decrypt it with the public key and store it in *ram*. Next, the software is hashed by the trusted code and that value is stored in *ram*. If the two hash values are matched, then the *computer* is assured that the software came from the *FACTORY* and it has not been altered. Based on that test, the software will be moved to *trustedMemory* or will remain in *untrustedMemory*.

[2]This list of public keys could be a keystore, TPM or hardcoded.

[3]The public key must be hashed along with the rest of the software.

**Secure Software Load Protocol Page**

Name: Secure Software Load
Benefits:

A. Based on trusted $code_\alpha$, establish the trustworthiness of $code_\beta$.

---

The goal of this protocol is: $code_\beta \in trustedMemory$ if the code is trustworthy and $code_\beta \in untrustedMemory$ if the code is not trustworthy.

**Notes**

- $code_\beta$ could be UEFI drivers, nvRam (such as “BIOS Configuration”), operating system loaders and the entire operating system.
- This model does not protect against replay attacks. If FACTORY signs and ships version 1, 2 and 3, any one of those versions may be loaded into the system. This can be resolved by using $TPM$s to store version information in $nvIndex$s protected by monotonically increasing version numbers. It could also be resolved by storing version information in $code_\alpha$.

**Secure Software Load Principal Page**

| **Principal** | **Description** |
|---|---|
| $CPU$ | The central processing unit of the computer. |
| $FACTORY$ | The factory where $privKey_\beta$ is securely stored and $code_\beta$ is signed. |

---

$CPU ::= \{\}$
$FACTORY ::= \{privKey_\beta, unsignedCode_\beta\}$

**Notes**

- We limit our model to a single CPU.
- In this model, the CPU is a leaf node.

**Secure Software Load Actor Page**

| **Actor** | **Description** |
|---|---|
| $world$ | The world. |
| $computer$ | All of the hardware and software in a given computer. |
| $memory$ | Memory that's available to the computer. |
| $untrustedMemory$ | Configurations under this actor should not be trusted. |
| $trustedMemory$ | Configurations under this actor have been trusted by the $computer$. |
| $code_\alpha$ | All of the memory (code, data, etc.) that has been trusted by the $computer$. |
| $ram$ | Random Access Memory. Generally untrusted unless first initialized by trusted code. |
| $privKey_\beta$ | The private key for $\beta$. It must be securely stored in $factory$. |

| **Leaf Node Actor** | **Description** |
|---|---|
| $trustedPubKeys$ | Either a list of trusted public keys OR code that will get the trusted public keys from another trusted source. For example, code to verify a certificate chain or a certificate kept in a TPM. |
| $code_\beta$ | The candidate code that is being evaluated for trustworthiness. This may not be code, it could be data. For example, the "BIOS Configuration" stored in nvRam could be $code_\beta$. |
| $pubKey_\beta$ | The public counterpart to $privKey_\beta$. |
| $hash_{\beta actual}$ | The computed hash of $code_\beta$ in $memory$. |
| $hash_{\beta decrypted}$ | The decrypted hash from $code_\beta$ in $memory$. |

---

$world ::= \{FACTORY, computer\}$

$computer ::= \{\underline{memory}, \underline{CPU}\}$

$memory ::= \{\underline{trustedCode}, \underline{untrustedCode}, \underline{ram}\}$

$trustedMemory ::= \{\underline{code_\alpha}\}$

$code_\alpha ::= \{\underline{trustedPubKeys}\}$

$privKey_\beta ::= \{\underline{pubKey_\beta}\}$

**Notes**

- *world* ties everything together... most of the axioms require a root to the Actor Network.
- In this model of asymmetric cryptography, the public key is derived from the private key.

**Secure Software Load Type Page**

| Type | Description |
|---|---|
| $\boldsymbol{software}$ | Any piece of software or data. |
| $\boldsymbol{pubKey}$ | The public portion of a public-private keypair. |
| $\boldsymbol{hash}$ | A cryptographically secure hash value := *hash*(*message*) |

---

$\boldsymbol{software} \rtimes \{code_\beta\}$
$\boldsymbol{pubKey} \rtimes \{pubKey_\beta\}$
$\boldsymbol{hash} \rtimes \{hash_{\beta actual}, hash_{\beta decrypted}\}$

---

**Notes**

- hash() is a cryptographic one-way function with a security property where it is prohibitively hard (due to size and computational complexity) for an attacker to deliberately generate a collision.

**Secure Software Load Configuration Page**

The initial configuration of this protocol is:

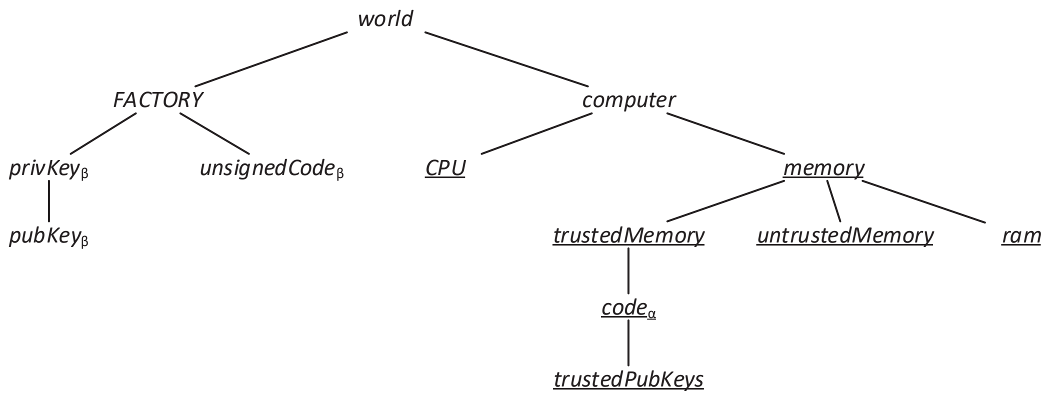
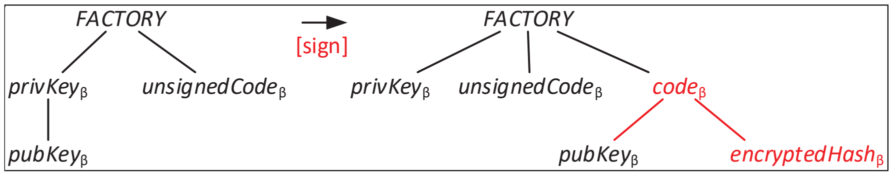


Figure 5.8: Initial Configuration of Software Load Protocol

In mathematical terms, the initial configuration is:

$world := \{FACTORY, computer\}$
$FACTORY := \{privKey_\beta, unsignedCode_\beta\}$
$privKey_\beta := \{pubKey_\beta\}$
$computer := \{\underline{CPU}, \underline{memory}\}$
$memory := \{\underline{trustedMemory}, \underline{untrustedMemory}, \underline{ram}\}$
$trustedMemory := \{\underline{trustedPubKeys}\}$

**Secure Software Load Action Page**

| **Type** | **Description** |
|---|---|
| [$sign$] | The factory will do three things to prepare $unsignedCode_\beta$ for release.<br>1) Hash $unsignedCode_\beta$ 2) encrypt the hash with $privKey_\beta$ 3) compose $code_\beta$:<br>$\{privKey_\beta, unsignedCode_\beta\} \xrightarrow{FACTORY} \{code_\beta \copyright unsignedCode_\beta,$<br>(cont.) $encryptedHash_\beta := encrypt(hash(code_\beta), privKey_\beta),$<br>(cont.) $code_\beta \oplus encryptedHash_\beta, privKey_\beta \ominus pubKey_\beta, code_\beta \oplus pubKey_\beta\}$<br> FACTORY<br>privKeyβ<br>unsignedCodeβ<br>pubKeyβ<br>[sign]<br>FACTORY<br>privKeyβ<br>unsignedCodeβ<br>codeβ<br>pubKeyβ<br>encryptedHashβ |
| [$ship$] | Send software into the world:<br>$\{\boldsymbol{software} \sqsubset FACTORY\} \xrightarrow[ship]{FACTORY} \{FACTORY \ominus \boldsymbol{software}, world \oplus \boldsymbol{software}\}$ |
| [$load$] | Load software into untrusted memory:<br>$\{\boldsymbol{software} \sqsubset world\} \xrightarrow[load]{CPU} \{world \ominus \boldsymbol{software}, untrustedMemory \oplus \boldsymbol{software}\}$ |
| [$checkPubKey$] | In $computer$, check a $\boldsymbol{pubKey}$ against $trustedPubKeys$:<br>$\{\boldsymbol{pubKey}, trustedPubKeys\} \xrightarrow[checkPubKey]{CPU} \{\boldsymbol{pubKey}$ is trusted$\}$ |
| [$hash$] | A secure hash function:<br>$\{\boldsymbol{software}\} \xrightarrow[hash]{CPU} \{\boldsymbol{hash_{software}} := hash(\boldsymbol{software}), ram \oplus \boldsymbol{hash_{software}}\}$ |
| [$decrypt$] | A secure asymmetric decryption of a $\boldsymbol{hash}$:<br>$\{\boldsymbol{hash_{encrypted}}, \boldsymbol{pubKey}\} \xrightarrow[decrypt]{CPU} \{$<br>(cont.) $\boldsymbol{hash_{decrypted}} := decrypt(\boldsymbol{hash_{encrypted}}, \boldsymbol{pubKey}),$<br>(cont.) $ram \oplus \boldsymbol{hash_{decrypted}}\}$ |
| [$trust$] | Trust some $\boldsymbol{software}$:<br>$\{\boldsymbol{software} \sqsubset untrustedMemory\} \xrightarrow[trust]{CPU} \{untrustedMemory \ominus \boldsymbol{software},$<br>(cont.) $trustedMemory \oplus \boldsymbol{software}\}$ |

**Notes**

- *hash*() is a cryptologically secure hash function such that collisions are impossible to force, thus providing assurance that when two hash values are the same, then the two inputs are also the same.
- encrypt(clearData, key) is a cryptologically secure asymmetric encryption algorithm such that the only way to successfully decrypt the result is with knowledge of the asymmetric key (of the pair) that was not used to encrypt it.

**Secure Software Load Policy Page**

The hash function must hash all of the $\boldsymbol{software}$:

$hash_{\beta actual} := hash(\forall code_\beta)$

(not modeled) $FACTORY\&CPU$ must $hash(\boldsymbol{software})$ in the same order.

---

The factory will sign and ship $code_\beta$:

$(true) \Rrightarrow (FACTORY) : \{privKey_\beta, unsignedCode_\beta\}[sign]code_\beta[ship]\{FACTORY \ominus code_\beta, world \oplus code_\beta\}$

---

Validate $code_\beta$:

$\{code_\beta \sqsubset world \land code_\beta \not\sqsubset computer\}[load]\{code_\beta \sqsubset untrustedMemory\}$

$(pubKey_\beta \sqsubset trustedPubKeys) \Rrightarrow \{pubKey_\beta, trustedPubKeys\}[checkPubKeys]pubKey_\beta$ is trusted.

$(pubKey_\beta$ is trusted $) \Rrightarrow \{encryptedHash_\beta, pubKey_\beta\}[decrypt]\{$

(cont.) $hash_{\beta decrypted} := decrypt(encryptredHash_\beta, pubKey_\beta), ram \oplus hash_{\beta decrypted}\}$

$(true) \Rrightarrow \{code_\beta\}[hash]\{hash_{\beta actual} := hash(code_\beta), ram \oplus hash_{\beta actual}\}$

$(hash_{\beta actual} == hash_{\beta decrypted}) \Rrightarrow \{code_\beta \sqsubset untrustedMemory\}[trust]\{untrustedMemory \ominus code_\beta,$

(cont.) $trustedMemory \oplus code_\beta\}$

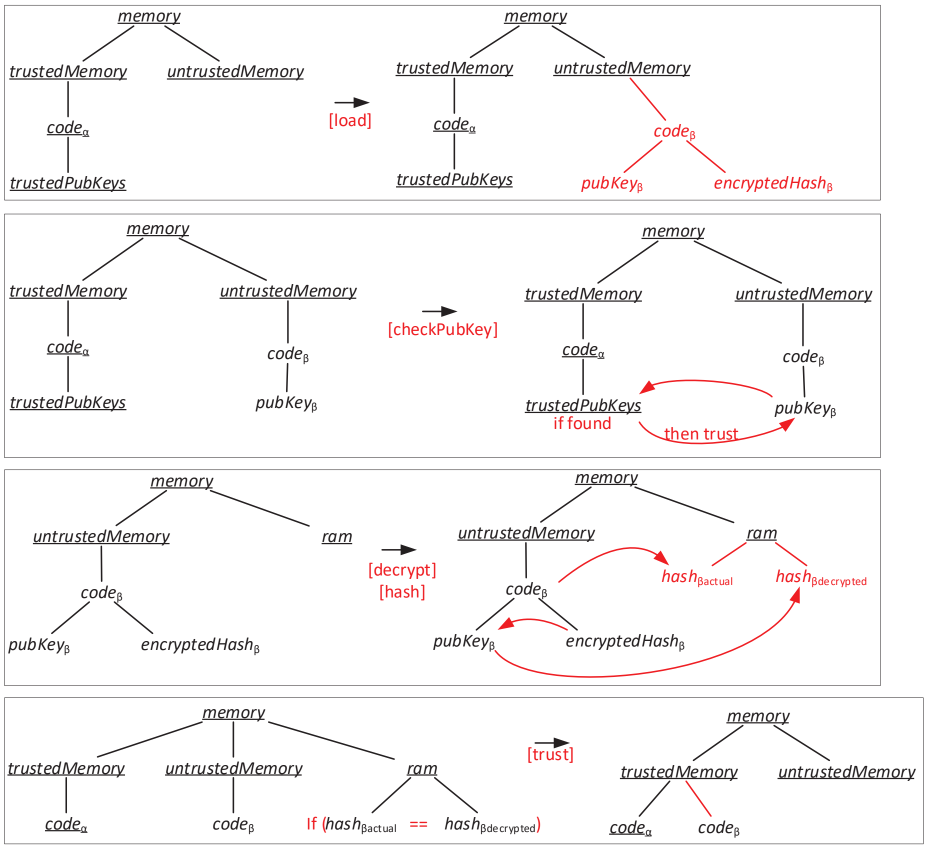


Figure 5.9: Software Load Protocol

**Secure Software Load Analysis Page**

**Analysis of what the network *should* do**

- The *computer* is able to load $code_{\beta}$ and trust it.
- If $code_{\alpha}$ does not trust $pubKey_{\beta}$ or if $code_{\beta}$ has been modified, then $code_{\beta}$ will not be trusted.

**Analysis of what the network *can* do**

- There are no protections around the two intermediate values: $hash_{\beta actual}$ and $hash_{\beta decrypted}$. This model makes no guarantees that these values are not altered after they are written but before they are read.
- This model does not protect against replay attacks. If FACTORY signs and ships version 1, 2 and 3, any one of those versions may be loaded into the system. This can be resolved by using $TPM$s to store version information in $nvIndex$s protected by monotonically increasing version numbers.

### 5.1.5 Secure UEFI Update Protocol

This protocol is, by far, the most complex. It builds on all of the concepts used in the earlier protocols and fully leverages the Secure Software Load protocol. Fortunately, the goal of this protocol is simple, it should load the new $\beta$ firmware into *nvRam* and update all of the nvIndex values (version and PCR) to ensure that the new firmware is trusted by the Secure UEFI Validate protocol.

In the Secure UEFI Validate protocol, the TPM had an object called nvIndex which was protected by a read policy. This policy ensured that the only way the TPM would allow access to the data stored in nvIndex (the version number of the UEFI firmware ) was if a PCR register matched a copy of a known-good PCR value securely stored in the nvIndex.

The model for this protocol is more complex because now, we need to work with two policies: An nvIndex WRITE policy and we need to compose, stage and eventually WRITE a new READ policy (and a new version number) for the nvIndex.

Before the protocol starts, the nvIndex must have been fully configured either in the $FACTORY$ or through a successful invocation of this protocol. Furthermore, there's a symmetric / shared secret password that is known only to the $ADMINISTRATOR$. This password is one of the controls (policies) that allows the nvIndex to be updated.

The model starts with a room containing three things, a *computer*, a $USER$ and an $ADMINISTRATOR$. The $ADMINISTRATOR$ knows the shared secret password, but the $USER$ does not.

The *computer* is also a little more complex as a *button* is added which can either be pressed or not pressed (see Equation 5.5).

Before we get into the protocol, we need to introduce a few new TPM concepts. The first concept is the idea of policies. This was first discussed in the Secure Software Load protocol, where a PCR register guarded access to a version number stored in an nvIndex. In this model, the nvIndex has two policies, a READ policy and a separate and more stringent WRITE policy. The read policy is the same policy we are familiar with from Secure Software Load. In this protocol, however, we will need to update, or WRITE new values into the nvIndex, which requires computing the PCR for the new firmware. The TPM, being a secure processor, has a lot of controls around updating certain properties of the TPM.

In order to boot the new $uefi_\beta$ firmware, we first need to update the $\boldsymbol{version}$ and $\boldsymbol{pcr}$ in the $TPM$ with new values. Otherwise, when the computer rebooted, the Secure UEFI Boot Protocol would detect that the firmware does not match the expected values in the $TPM$ and the firmware would not be trusted.

The new PCR is computed using a TPM object called a "Trial Session". A trial session is created using a dummy username (called the NULL hierarchy in TPM-nomenclature[4]). If the new firmware is trusted, then a Trial Session is created on the TPM and a PCR value is computed for the new firmware and staged in *ram*.

Next, another session called an "Update Session" must login with a shared secret. If the shared secret, for example, the one known by the $ADMINISTRATOR$ matches the secret in nvIndex, then the Update

[4]The concept of a user and password is called a 'hierarchy' in TPM-speak. When you login to a hierarchy, you create a new session. A NULL hierarchy is so named because it does not have a password. NULL-sessions are used to manage concurrent access to the TPM.

Session has unlocked one of several policy controls that must be satisfied in order to WRITE to the nvIndex.

Another TPM policy control required to update the nvIndex, is Physical Presence (PP). This TPM term is misnamed as the TPM specification clearly spells out methods to assert Physical Presence electronically! However, in our model we will use a *button* attached to the *computer* as the only way to assert Physical Presence. Initially, the button is not pressed. Our model says any person can press the button (not just the $ADMINISTRATOR$), however, the person must be in the same room as the computer.

The first step of the protocol is to load the new firmware into $trustedMemory$. Equation 5.6 uses the Secure Software Load protocol to bring the new firmware in.

The next step is to compute the new PCR value for the firmware. A Trial Session is created in the TPM and a (the model uses $\alpha$ for original firmware and $\beta$ for new firmware) PCR value is computed and staged in *ram* for the new firmware. Equation 5.7 leaves us in this state.

At this point the protocol tries to create an Update Session. If it's successful, then Equation 5.8 will either update the firmware or consign it to $untrustedMemory$. The first line of the equation states that the new firmware must be in trusted memory and the TPM does not have an active Update Session. The next assertion ensures that the shared secret passwords match and the $ADMINISTRATOR$ is now logged into the TPM with an Update Session. The final two authorizations are verifying that the button is pressed and the new version of the software is greater than the prior version. This protects against replay attacks or malicious actors loading trusted firmware with known vulnerabilities. If both of those controls are satisfied, then the update is authorized.

In summary, there are four controls protecting WRITE access to the nvIndex:

- The new firmware must be digitally signed by a secret key that is trusted by this *computer*.
- Knowledge of a secret password.
- Physical presence in the same room as the computer.
- The new version of firmware must be > than the current version.

When the update is authorized, three things must happen. Two values in the TPM must be updated: The new PCR value and the new version number. With the update fully authorized, the TPM is now allowed to write them into the nvIndex. Third, the actual firmware needs to be updated. In this case, we use operators from Table 4.5 to replace the old firmware with the new firmware.

If, for any reason, the update is not authorized, then the new firmware is moved to untrusted memory.

The full process to update the $TPM$ is very elaborate. This model attempts to simplify it into accurate, but easily modelable terms that will ultimately satisfy the goals of the protocol.

**Secure UEFI Update Protocol Page**

Name: Secure UEFI Update
Benefits:

A. Securely update the UEFI software. The update should have the following properties:
   - The update should be verified to be authentic code from the $FACTORY$ (modeled in Secure Software Load).
   - The update should be authorized with a shared secret $\boldsymbol{password}$ and the physical presence of a $\boldsymbol{person}$.
   - The update should be a newer, more up-to-date $\boldsymbol{version}$, not a $\boldsymbol{version}$ that is earlier than the current UEFI $\boldsymbol{firmware}$.

---

In plain language, the goal of this protocol is to update UEFI version $\alpha$ with version $\beta$. In mathematical terms, the goal of this protocol is: $(uefi_\alpha \notin nvRam) \wedge (uefi_\beta \in nvRam) \wedge (version_{nvIndex0\beta} \in nvIndex) \wedge (pcr0_{nvIndex\beta} \in pcr_0)$ if the code is trustworthy and authorized by an $ADMINISTRATOR$ or $uefi_\beta \in untrustedMemory$ if the code is not trustworthy.

**Notes**

- This model would allow $uefi_\beta$ to run when it's loaded into $trustedMemory$ during the Secure Software Load protocol. This should not be an issue because all of the code that is running is trusted.

## Secure UEFI Update Principal Page

| Principal | Description |
|---|---|
| $ADMINISTRATOR$ | A $\boldsymbol{person}$ who can authorize the installation of new $\boldsymbol{firmware}$. |
| $CPU$ | The central processing unit of the *computer*. |
| $TPM$ | Trusted platform module. A secure storage/crypto processor that is tamper resistant. |
| $USER$ | A $\boldsymbol{person}$ who is not authorized to install $\boldsymbol{firmware}$. |

---

$ADMINISTRATOR ::= \{adminPW_{admin}\}$

$CPU ::= \{\underline{pc}\}$

$TPM ::= \{\boldsymbol{session}, nvIndex, pcr_0\}$

$USER ::= \{\}$

### Notes

- We limit our model to a single CPU.
- $USER$, who does not have the *adminPW* is intended to contrast with the $ADMINISTRATOR$ who does have it. Both can assert physical presence, but only one has a secret that is shared with the $TPM$.

**Secure UEFI Update Actor Page**

| Actor | Description |
|---|---|
| $room$ | A location that contains a $\boldsymbol{person}$ and a $computer$. |
| $computer$ | All of the hardware and software in a given $computer$. |
| $button$ | A physical presence sensor somewhere in the $computer$. |
| $memory$ | Memory that's available to the $computer$. |
| $trustedMemory$ | Configurations under this actor have been trusted by the $computer$. |
| $untrustedMemory$ | Configurations under this actor should not be trusted. |
| $nvRam$ | A trusted, nonvolatile block of electronically reprogrammable memory. |
| $uefi_{\alpha}$ | Current version of UEFI $\boldsymbol{firmware}$ that is trusted by the $computer$. |
| $uefi_{\beta}$ | New version of UEFI $\boldsymbol{firmware}$ that will be loaded and trusted by the $computer$ using the Secure Software Load Protocol. |
| $pcr_0$ | Core root of trust measurement (CRTM). It contains the measurement of the BIOS / UEFI firmware. |
| $trialSession$ | A non-binding $TPM$ session that is used to compute $\boldsymbol{pcr}$ values. |
| $updateSession$ | An authenticated $TPM$ session that can update $\boldsymbol{nvIndex}$ values. |
| $ram$ | Random Access Memory. Generally untrusted unless first initialized by trusted code. |
| $nvIndex_0$ | Contains a $\boldsymbol{version}$ of the BIOS / UEFI $\boldsymbol{firmware}$ and a copy of $pcr_0$ at the time the BIOS / UEFI firmware was trusted. Read access is protected by $pcr_0$. Write access is protected by a more complex policy. |

| Leaf Node Actor | Description |
|---|---|
| $adminPW_{admin}$ | A shared secret known by both the $ADMINISTRATOR$ and a $nvIndex$ in the $TPM$. |
| $adminPW_{nvIndex0}$ | A shared secret in the $TPM$ that protects $nvIndex_0$ from writing. |
| $buttonNotPressed$ | Physical presence is not asserted. |
| $buttonPressed$ | Physical presence by a $\boldsymbol{person}$ is asserted. |
| $pc$ | Holds the Program Counter or a pointer to the current instruction. The model will assert that this always points to addresses that are part of trusted memory, although this won't be explicitly modeled. Proof of correctness is in earlier models. |
| $pcr_{0nvIndex\beta}$ | The precomputed value of $pcr_0$ when $uefi_{\beta}$ has been extended into it. |
| $reset$ | The uninitialized state of an actor. |
| $version_{nvIndex0\alpha}$ | The UEFI version number securely stored in $nvInxex_0$. |
| $version_{nvIndex0\beta}$ | A copy of the UEFI version number stored in $uefi_{\beta}$. |
| $version_{uefi\alpha}$ | The UEFI version number stored in $uefi_{\alpha}$. |
| $version_{uefi\beta}$ | The UEFI version number stored in $uefi_{\beta}$. |

---

$room ::= \{USER, ADMINISTRATOR, computer\}$

$computer ::= \{\underline{button}, \underline{CPU}, \underline{memory}, \underline{TPM}\}$

$button ::= \{\underline{\boldsymbol{buttonState}}\}$
$memory ::= \{\underline{trustedMemory}, \underline{untrustedMemory}, \underline{ram}\}$
$trustedMemory ::= \{\underline{nvRam}, \boldsymbol{firmware}\}$
$untrustedMemory ::= \{\{\}, \boldsymbol{firmware}\}$
$nvRam ::= \{\boldsymbol{firmware}\}$
$uefi_{\alpha} ::= \{\underline{version_{uefi\alpha}}\}$
$uefi_{\beta} ::= \{\underline{version_{uefi\beta}}\}$
$nvIndex_0 ::= \{\underline{\boldsymbol{version}}, \underline{\boldsymbol{pcr}}, \underline{\boldsymbol{password}}\}$
$trialSession_0 ::= \{\boldsymbol{pcr}\}$

**Notes**

- Some actors may be missing as they are very short lived and can easily be tracked between states and then no longer used.

**Secure UEFI Update Type Page**

| **Type** | **Description** |
|---|---|
| $\boldsymbol{buttonState}$ | The state of a button on the *computer*. |
| $\boldsymbol{firmware}$ | Software that's been released by a FACTORY and used as UEFI firmware in a *computer*. |
| $\boldsymbol{hash}$ | A cryptographically secure hash value := $hash(message)$ |
| $\boldsymbol{nvIndex}$ | Non-volatile RAM in the TPM that is protected by a read policy and a different write policy. |
| $\boldsymbol{password}$ | A shared secret that should only be known by an $ADMINISTRATOR$. It authenticates a $\boldsymbol{person}$ for access to $\boldsymbol{nvIndexes}$ in the $TPM$. |
| $\boldsymbol{pcr}$ | Platform Configuration Register. A special memory cell in the $TPM$ that can not be directly set by anything outside of the TPM. Instead, it's value is calculated as: $tpm_{old} := hash(tpm_{old} \| tpm_{new})$. |
| $\boldsymbol{person}$ | A person who may have a $\boldsymbol{password}$ and may be able to push a *button*. |
| $\boldsymbol{session}$ | A session which is used to create or access objects in the $TPM$ depending on what is authorized. |
| $\boldsymbol{version}$ | The version number of something the protocol is attempting to trust. |

---

$\boldsymbol{buttonState} \rtimes \{buttonNotPressed, buttonPressed\}$

$\boldsymbol{firmware} \rtimes \{uefi_\alpha, uefi_\beta\}$

$\boldsymbol{hash} \rtimes \{pcr_{0uefi\alpha}, pcr_{0uefi\beta}\}$

$\boldsymbol{nvIndex} \rtimes \{nvIndex_0\}$

$\boldsymbol{password} \rtimes \{adminPW_{admin}, adminPW_{nvIndex0}\}$

$\boldsymbol{pcr} \rtimes \{pcr_0\}$

$\boldsymbol{person} \rtimes \{USER, ADMINISTRATOR\}$

$\boldsymbol{session} \rtimes \{trialSession, updateSession\}$

$\boldsymbol{version} \rtimes \{version_{uefi\alpha}, version_{uefi\beta}, version_{nvIndex0\alpha}, version_{nvIndex0\beta}\}$

---

$\boldsymbol{firmware} ::= \{\underline{\boldsymbol{version}}\}$

$\boldsymbol{nvIndex} ::= \{\underline{\boldsymbol{version}}, \underline{\boldsymbol{pcr}}, \underline{\boldsymbol{password}}\}$

$\boldsymbol{pcr} ::= \{\underline{\boldsymbol{hash}}\}$

$\boldsymbol{session} ::= \{\underline{\boldsymbol{password}}, \boldsymbol{hash}\}$

---

**Notes**

- hash() is a cryptographic one-way function with a security property where it is prohibitively hard (due to size and computational complexity) for an attacker to deliberately generate a collision.

## Secure UEFI Update Configuration Page

The initial configuration of this protocol is:

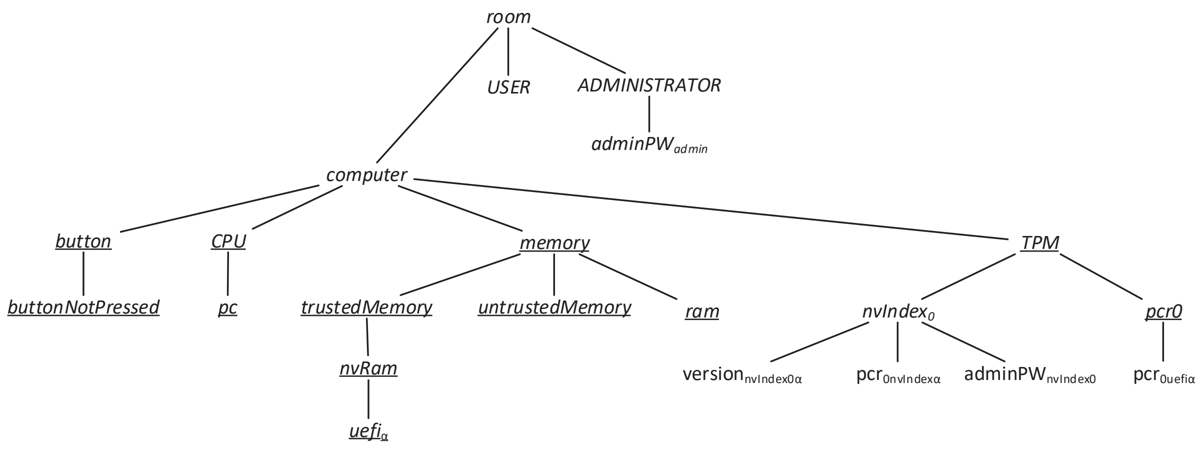


Figure 5.10: Initial Configuration of the UEFI Update Protocol

In mathematical terms, the initial configuration is:

$room := \{USER, ADMINISTRATOR, computer\}$

$ADMINISTRATOR := \{adminPW_{admin}\}$

$computer := \{\underline{button}, \underline{CPU}, \underline{memory}, \underline{TPM}\}$

$button := \{buttonNotPressed\}$

$CPU := \{pc\}$

$memory := \{\underline{trustedMemory}, \underline{untrustedMemory}, \underline{ram}\}$

$trustedMemory := \{\underline{nvRam}\}$

$nvRam := \{uefi_{\alpha}\}$

$uefi_{\alpha} := \{\underline{version_{uefi\alpha}}\}$

$uefi_{\beta} := \{\underline{version_{uefi\beta}}\}$

$TPM := \{nvIndex_0, \underline{pcr_0}\}$

$nvIndex_0 := \{version_{nvIndex0\alpha}, pcr_{0nvIndex\alpha}, adminPW_{nvIndex0}\}$

$version_{uefi\alpha} == version_{nvIndex0\alpha}$

$pcr_{0nvIndex\alpha} == pcr_{0uefi\alpha}$

$pcr_0 := \{pcr_{0uefi\alpha}\}$

**Secure UEFI Update Action Page**

| Type | Description |
|---|---|
| [*createTrialSession*]<br><br>See Figure 5.11 | Create a $\boldsymbol{session}_{trial}$ in the $TPM$ that will be used later to calculate a new $\boldsymbol{pcr}$ value:<br><br>$\{\boldsymbol{session}_{trial} \not\sqsubset TPM\} \xrightarrow[createTrialSession]{CPU,TPM}$<br>(cont.) $\{\boldsymbol{session}_{trial} := newTrialSession(NULL_{password}),$<br>(cont.) $TPM \oplus \boldsymbol{session}_{trial},$<br>(cont.) $\boldsymbol{session}_{trial} \oplus \boldsymbol{pcr}_{trial},$<br>(cont.) $\boldsymbol{pcr}_{trial} \oplus reset\}$ |
| [*extendPCR*]<br><br>See Figure 5.12 | Using the $\boldsymbol{session}_{trial}$ and the new $\boldsymbol{firmware}$, compute a PCR hash for it:<br><br>$\{\boldsymbol{session}_{trial} \in TPM \wedge \boldsymbol{firmware}_{new} \sqsubset memory\} \xrightarrow[extendPCR]{CPU,TPM}$<br>(cont.) $\{\boldsymbol{hash} := hash(\boldsymbol{firmware}_{new}),$<br>(cont.) $\boldsymbol{pcr}_{trial} \ominus reset, \boldsymbol{pcr}_{trial} \oplus \boldsymbol{hash}\}$ |
| [*getPolicy*]<br><br>See Figure 5.13 | Copy/move the computed pcr (it's a $\boldsymbol{hash}$) to *ram*:<br><br>$\{\boldsymbol{hash} \sqsubset \boldsymbol{pcr}_{trial}\} \xrightarrow[getPolicy]{CPU,TPM} \{\boldsymbol{pcr}_{trial} \ominus \boldsymbol{hash}, ram \oplus \boldsymbol{hash}\}$ |
| [*createUpdateSession*]<br><br>See Figure 5.14 | Using the password, create a session that is authorized to update firmware (this is guarded by a password check):<br><br>$\{\boldsymbol{session}_{update} \not\sqsubset TPM\} \xrightarrow[createTrialSession]{CPU,TPM}$<br>(cont.) $\{\boldsymbol{session}_{update} := newUpdateSession(\boldsymbol{password}),$<br>(cont.) $TPM \oplus \boldsymbol{session}_{update}\}$ |
| [*authorizeUpdate*] | Use physical presence to authorize the firmware update:<br><br>$\{buttonPressed \in button \wedge \boldsymbol{session}_{update} \sqsubset TPM\} \xrightarrow[authorizeUpdate]{\boldsymbol{person}}$<br>(cont.) $\{updateAuthorized\}$ |
| [*writePolicy*]<br><br>See Figure 5.15 | Using an authorized session, update the nvIndex:<br><br>$\{updateAuthorized\} \xrightarrow[writePolicy]{TPM} \{ram \ominus \boldsymbol{hash}, \boldsymbol{nvIndex} \oplus \boldsymbol{hash},$<br>(cont.) $\boldsymbol{nvIndex} \ominus \boldsymbol{version}_{old}, \boldsymbol{nvIndex} \oplus \boldsymbol{version}_{new}\}$ |
| [*saveFirmware*]<br><br>See Figure 5.16 | Move the new firmware into trusted nvRam, replacing the old firmware:<br><br>$\{updateAuthorized\} \xrightarrow[saveUefi]{CPU} \{nvRam \ominus \boldsymbol{firmware}_{old},$<br>(cont.) $nvRam \oplus \boldsymbol{firmware}_{new}\}$ |

| Type | Description |
|---|---|
| $[rejectFirmware]$<br><br>See Figure 5.17 | Move the new firmware into untrustedMemory:<br><br>$\{\neg updateAuthorized\} \xrightarrow[rejectFirmware]{CPU} \{trustedMemory \ominus \boldsymbol{firmware}_{new},$<br>(cont.) $untrustedMemory \oplus \boldsymbol{firmware}_{new}\}$ |
| $[buttonPress]$ | A $\boldsymbol{person}$ pushes a $button$:<br><br>$\{buttonNotPressed \in button\} \xrightarrow[buttonPress]{\boldsymbol{person}} \{button \ominus buttonNotPressed,$<br>(cont.) $button \oplus buttonPressed\}$ |
| $[buttonRelease]$ | A $\boldsymbol{person}$ releases a $button$:<br><br>$\{buttonPressed \in button\} \xrightarrow[buttonRelease]{\boldsymbol{person}} \{button \ominus buttonPressed,$<br>(cont.) $button \oplus buttonNotPressed\}$ |

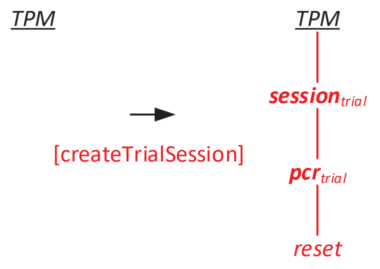


Figure 5.11: Create a Trial Session

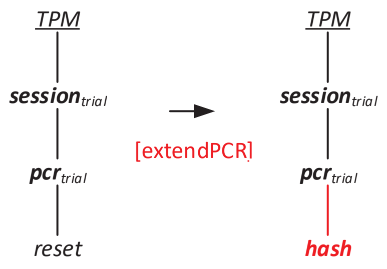


Figure 5.12: Extend the PCR Register

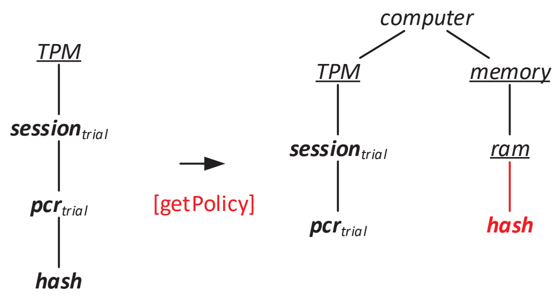


Figure 5.13: Save the New PCR Value to Memory

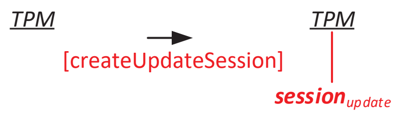


Figure 5.14: Create an Update Session

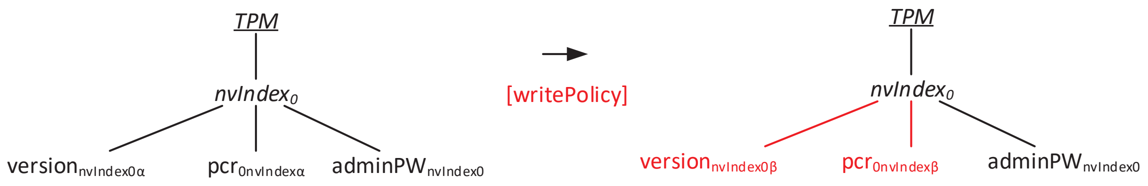


Figure 5.15: Writing a New TPM Policy to an nvIndex

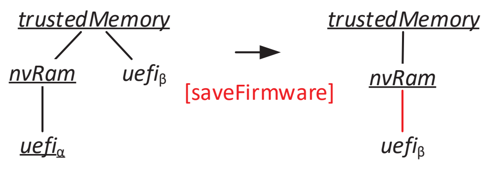


Figure 5.16: Saving the Firmware Update

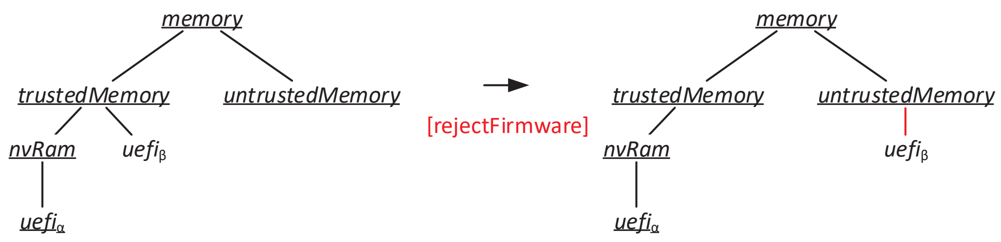


Figure 5.17: Rejecting the Firmware Update

**Notes**

- $newTrialSession()$ and $newUpdateSession()$ are $TPM$ functions that create a new session. All sessions require a password, but Trial sessions use a NULL password.

### Secure UEFI Update Policy Page

Any $\boldsymbol{person}$ can push the *button* but only one $\boldsymbol{person}$ can press it at a time:

$$\begin{aligned}(buttonNotPressed \in button) \Rrightarrow (\boldsymbol{person}) : [buttonPress]\\ (buttonPressed \in button) \Rrightarrow (\boldsymbol{person}) : [buttonRelease]\end{aligned} \tag{5.5}$$

---

A new version of $uefi_\beta$ will be loaded with the Secure Software Load Protocol:

$$(uefi_\beta \not\sqsubset memory) \Rrightarrow (\boldsymbol{person}) : [SecureSoftwareLoad](uefi_\beta) \tag{5.6}$$

If it's trusted, then the resultant state is: $\{uefi_\beta \sqsubset trustedMemory\}$

---

Compute a $\boldsymbol{pcr}$ value for $uefi_\beta$:

$$\begin{aligned}(trialSession \not\sqsubset TPM) \Rrightarrow (CPU) : [createTrialSession](NULL)\, \{trialSession \sqsubset TPM \wedge\\ (cont.) \qquad reset \in pcr_{trial}\}\\ (trialSession \sqsubset TPM) \Rrightarrow (CPU) : [extendPCR](uefi_\beta)\, \{pcr_{0nvIndex\beta} \in pcr_{trial}\}\\ \{(trialSession \sqsubset TPM) \Rrightarrow (CPU) : [getPolicy](trialSession)\, \{pcr_{0nvIndex\beta} \in ram\}\end{aligned} \tag{5.7}$$

---

Update the UEFI code:

$$\begin{aligned}(uefi_\beta \sqsubset trustedMemory) \wedge (updateSession \not\sqsubset TPM) \wedge (adminPW_{admin} == adminPW_{nvIndex0})\\ \Rrightarrow (ADMINISTRATOR) : [createUpdateSession](adminPW_{admin})\, \{updateSession \sqsubset TPM\}\\ (buttonPressed \in button) \wedge (version_{uefi\beta} > version_{uefi\alpha}) \Rrightarrow (ADMINISTRATOR) :\\ (cont.) \qquad [authorizeUpdate]\{updateAuthorized\}\\ (updateAuthorized) \Rrightarrow (TPM) : [writePolicy](pcr_{0nvIndex\beta}, version_{nvIndex0\beta})\\ (cont.) \qquad \{pcr_{0nvIndex\beta} \in nvIndex_0 \wedge version_{nvIndex0\beta} \in nvIndex_0\}\\ (updateAuthorized) \Rrightarrow (CPU) : [saveFirmware](uefi_\beta)\, \{uefi_\beta \in nvRam \wedge uefi_\alpha \notin nvRam\}\\ (\neg updateAuthorized) \Rrightarrow (CPU) : [rejectFirmware](uefi_\beta)\, \{uefi_\beta \in untrustedMemory \wedge\\ (cont.) \qquad uefi_\alpha \in nvRam\}\end{aligned} \tag{5.8}$$

**Secure UEFI Update Analysis Page**

**Analysis of what the network *should* do**

- If the $ADMINISTRATOR$ uses it's $adminPW_{admin}$, the $uefi_{\beta}$ firmware is trusted, the version of $\beta > \alpha$ and someone presses the *button* at the right time, then the $uefi_{\alpha}$ firmware will be replaced with $uefi_{\beta}$.
- If any of the above conditions are not true, then $uefi_{\beta}$ should be moved to $untrustedMemory$.

**Analysis of what the network *can* do**

- There are no protections around the $pcr_{0nvIndex\beta}$ hash value that's stored in *ram*. This model makes no guarantees that this value is not altered after it is written but before it's read.
- There are no protections around the data as it flows between *memory* and the $TPM$. It could get altered in transit.
- There are several opportunities for race conditions in this protocol and it is not safe to run concurrent executions of it. Some type of locking mechanism is recommended, but is not modeled.
- The protocol does not force the *computer* to reboot. If shadow ram is used, then the old $uefi_{\alpha}$ code could run indefinitely even though the protocol's goals have been satisfied ($uefi_{\beta} \in nvRam$).

# CHAPTER 6
# CONCLUSIONS

Don't let the noise of others' opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition.

–Steve Jobs, *Stanford University Commencement Address, 2005*

This thesis covered a number of topics relevant to Computer Science. It began with a simple question: "Why do we adopt protocols that are nominally well-behaved but are unsafe and insecure?" It observes that one reason this problem continues to plague us is because companies can get away with it. Publishing insecure systems does not yet carry enough of a penalty for rational people to choose to invest in security.

Computer Science is not the first discipline to grapple with these issues. We analyzed two industries that were also the product of fundamental scientific research and was engineered for commercial and retail use by the general public. These industries, Aviation and Urban Safety both have an inherent potential danger but also add tremendous value to society. They are worthy of study because, clearly, they are doing something right.

We need to study software failures the same way we analyze airline accidents and urban disasters. We must acknowledge that these things will happen and take steps to mitigate the damage when it does. Regulation, from the government or through private bodies like insurers, must be able to scale and be, above all, practical and easy to implement. In the long run, it must pay for itself. We need to systematically measure the performance of the regulation and the utility of the technology to the general public to make the case that a small investment in safety and security will ultimately benefit them.

The analysis of each industry identified 11 lessons that can be applied to Cyber Security. The Urban Safety laid out an even stronger methodology for improving safety of systems across a diverse set of disciplines. It concluded with the postulate that: If software manufacturers must pay a proportional cost when they produce a defective product, then a beneficial ecosystem will develop.

One key ingredient for the safety in both industries is the capability to detect and analyze failures rigorously. Recent security attacks try to circumvent detection by hiding in hardware devices and by gaining control before a system is fully initialized. For these attacks the detection of the security threats are becoming challenging and any risk model becomes harder to apply. These attacks have motivated the development of secure boot mechanisms and secure memory enclaves (like INTEL SGX) that can provide privacy from the overarching system. Both mechanisms rely on features of the processor hardware and physical interaction from the user. Their proper operation needs to be modeled as a cyber-physical system. Our proposed methodologies improve our capabilities to detect security attacks and to model and analyze security protocols for cyber-physical systems.

The thesis then introduced a formal methodology called Actor Network Modeling. The goal of Actor Network Theory is to use scientific methods to model protocols and analyze their performance against goals. The thesis draws from several papers from both Sociology and Computer Science and synthesizes all of this material to produce a detailed methodology for creating Actor Network Models. It describes a common set of terminology, notation and useful axioms. Section 4.5 borrows from the waterfall software development methodology and introduces the concept of organizing the models into a set of 8 ‘pages’ that fully describe a protocol. The chapter concludes with a fully working, non-technical example and proof.

Finally, the thesis uses this methodology to analyze the functionality and security requirements of a real-world set of protocols collectively known as Secure Boot by modeling 4 interdependent protocols and analyzing various security properties. The analysis identifies weaknesses in the protocols such as the fact that the security measurements are not continuous and that trust may erode over time. Oftentimes, there is no protection for intermediate values stored in ram, which is readily accessible by other technologies such as Direct Memory Access. Finally, the analysis points out that there are issues with revocation of trust, lack of protection from replay attacks and a potential for race conditions.

## 6.1 Future Research

The proposed methodology has several restrictions. This work tries to strike a balance between a strict formal declaration (like a computer language) that is clear and type-safe with the spirit of Actor Network Theory[31] that eschews formalism. The more models we build with this methodology, the more experience we will develop on how to evolve it.

In particular, the proposed notation lacks a method for identifying cardinality of membership. For example, if actor A may contain b, c, d and e; can it choose none, one, two or more, etc. This would be helpful for someone who approaches modeling like programming.

Extend the methodology to include a threat model.

Finally, an attempt could be made to use this modeling technique to formally verify aspects of the full Secure Boot protocol rather than simply analyze a few properties.

# APPENDIX A
# GLOSSARY

Table A.1: Acronyms used in this paper.

| Acronym | Definition |
|---|---:|
| ALSR | Address Space Layout Randomization |
| ANT | Actor Network Theory |
| API | Application Program Interface |
| BIOS | Basic Input/Output System |
| CRTM | Core Root of Trust Measurement |
| DoS | Denial of Service |
| EULA | End User License Agreement |
| HIPAA | Health Insurance Portability and Accountability Act of 1996 |
| IETF | Internet Engineering Task Force |
| IP | Internet Protocol |
| MAC | Media Access Control |
| NBFU | National Board of Fire Underwriters |
| NEC | National Electrical Code |
| NFPA | National Fire Protection Association |
| NIST | National Institute of Standards and Technology |
| PCI | Payment Card Industry |
| PDL | Procedure Derivation Logic |
| PDL | Protocol Derivation Logic |
| RFC | Request for Comment |
| SGX | Software Guard Extensions |
| TCB | Trusted Computing Base |
| TCG | Trusted Computing Group |
| TPM | Trusted Platform Module |
| TXT | Trusted Execution Technology |
| UEFI | Unified Extensible Firmware Interface |
| UL | Underwriter Laboratories |

# APPENDIX B
# AXIOMS USED FROM EARLIER WORKS

This thesis depends on a number of axioms that were defined in earlier works.

Table B.1: The Axioms Used in this Thesis

| **Paper** | **Axiom** | **Description** |
|---|---|---|
| [34] | Definition 1 | A set (née configuration) is a finite set. It may be empty or it may contain actors or other configurations. |
| | Definition 2 | A configuration (née box configuration) is a hierarchy of configurations. A configuration may be empty, contain actors or other configurations. |
| | Definition 3 | Defines membership within the hierarchy. For each actor in the hierarchy, it's either the entire hierarchy or all its parents 'contain' it. |
| | Definition 4 Proposition 1 | Each actor in a configuration (organized as a hierarchy) must be unique. |
| | Definition 5 | $\sigma$ (née *mult*) is used to count the number of occurrences of an actor in a configuration. |
| | Proposition 2 | If $X \sqsubseteq Y$ then $\sigma(X, \boldsymbol{t}) \leq \sigma(Y, \boldsymbol{t})$ |
| | Definition 6 | Defines a transaction where an actor moves from one configuration to another. |
| | Definition 7 | When an actor moves to a new configuration, it takes everything under its hierarchy with it. |
| | Definition 8 | Describes the notation of a transaction and weather it's a unique action or one of a possible set of actions. |
| | Move types 1-6 in section 3.1 | Describes the nomenclature for how principles direct the action of actors. |
| | Definition 9 | Defines a run as an alternating sequence of states and moves. |
| | Definition 10 | Defines the notion of a policy and how it guards a set of runs. It's the logical equivalent of what the runs *can* do (the power set of possible runs) and what they *should* do (the policy). |
| | Definition 11 | Defines the notion of a move policy – under what conditions an action may take place. |
| [40] | Easy Subterms | If a principle knows an actor, it also knows all actors that are subsets of it. |
| | Section 3.3.1 & Section 4.3.1 | The notion of sending/receiving events. |
| | Section 4.3.1 | The notion of a freshly generated nonce. |

# BIBLIOGRAPHY

[1] *W. Edwards Deming Quotes*.
`http://quotes.deming.org/10084`

[2] *The Great Storm*. *The Cambridge Press*, XXII(50) (March 1888).
`http://cambridge.dlconsulting.com/cgi-bin/cambridge?a=d&d=Press18880317-01.2.16`

[3] *The Men who Made the NFPA*. In *NFPA Journal* (1996).

[4] *Set theory: difference between belong/contained and includes/subset?* (2012).
`https://math.stackexchange.com/questions/131309/set-theory-difference-between-belong-contained-and-includes-subset`

[5] *History of NFPA* (2017).
`http://www.nfpa.org/about-nfpa/nfpa-overview/history-of-nfpa`

[6] *IETF document statistics (all RFCs)* (2017).
`http://www.arkko.com/tools/rfcstats/`

[7] *Largest Fire Losses in the United States* (2017).
`http://www.nfpa.org/news-and-research/fire-statistics-and-reports/fire-statistics/fires-in-the-us/large-property-loss/largest-fire-losses-in-the-united-states`

[8] ACM. *Curriculum Guidelines for Undergraduate Degree Programs in Cybersecurity*. Association for Information Systems Special Interest Group on Security (AIS SIGSEC). Association of Computing Machinery, version 0.5 edition (January 2017).

[9] Albrechtsen, Eirik. *Security vs safety* (2003).

[10] Arthur, Will; Challener, David and Goldman, Kenneth. *A Practical Guide to TPM 2.0*. Apress (2015).

[11] Atmel. *System Design Manufacturing Recommendations for Atmel TPM Devices* (2013).

[12] Beck, Ulrich. *Risk Society: Towards a New Modernity*. SAGE Publications (1992).

[13] Bottomley, James and Corbet, Jonathan. *Making UEFI secure boot work with open platforms*. *The Linux Foundation*, 49 (2011).

[14] Bulygin, Yuriy; Furtak, Andrew and Bazhaniuk, Oleksandr. *A Tale of One Software Bypass of Windows 8 Secure Boot*. *Black Hat* (2013).

[15] Butterworth, John et al. *BIOS Chronomancy: Fixing the Core Root of Trust for Measurement*. In *ACM SIGSAC Conf. on Comp. & Comm. Security*, (pp. 25–36) (2013). ISBN 978-1-4503-2477-9.

[16] Callon, Michel. *The Sociology of an Actor-Network: The Case of the Electric Vehicle*, (pp. 19–34). Palgrave Macmillan UK, London (1986). ISBN 978-1-349-07408-2.
`http://dx.doi.org/10.1007/978-1-349-07408-2_2`

[17] Cooper, David et al. *BIOS Protection Guidelines*. Technical Report 800-147, NIST (2011).

[18] Cormen, Thomas H et al. *Introduction to Algorithms*. MIT Press, Cambridge, MA (1990).

[19] Corp., Microsoft. *Microsoft .NET Framework Redistributable EULA* (2017).
`https://msdn.microsoft.com/en-us/library/ms994405.aspx`

[20] Embleton, Shawn; Sparks, Sherri and Zou, Cliff. *SMM Rootkits: A New Breed of OS Independent Malware*. *ACM* (2008).
`http://www.eecs.ucf.edu/~czou/research/SMM-Rootkits-Securecom08.pdf`

[21] Group, The World Bank. *Air transport, passengers carried*.
`http://data.worldbank.org/indicator/IS.AIR.PSGR`

[22] Hoare, Charles Antony Richard. *An axiomatic basis for computer programming*. *Communications of the ACM*, 12(10):pp. 576–580 (1969).

[23] Hoare, Charles Antony Richard. *The Emperor's Old Clothes*. *Communications of the ACM*, 24(2):pp. 75–83 (February 1981). ISSN 0001-0782.
`http://doi.acm.org/10.1145/358549.358561`

[24] Infineon. *Infineon SLB 9645 TPM with Embedded Platform* (October 2014).

[25] Intel. *Intel Trusted Platform Module (TPM module-AXXTPME3) Hardware User's Guide*. Technical Report G21682-003, Intel Corporation (2011).

[26] Intel. *Intel Trusted Execution Technology (Intel TXT) - Software Development Guide* (July 2015).

[27] Jonnes, Jill. *Empires of light: Edison, Tesla, Westinghouse, and the race to electrify the world*. Random House Trade Paperbacks (2003).

[28] Kallenberg, C et al. *Extreme privilege escalation on UEFI Windows 8 systems*. In *Hack.LU* (2014).

[29] Knowles, Scott Gabriel. *The Disaster Experts: Mastering Risk in Modern America*. University of Pennsylvania Press, Philadelphia, Pennsylvania 19104-4112 (2011). ISBN 978-0-8122-4350-5.

[30] Komons, Nick A. *Bonfires to Beacons: Federal Civial Aviation Policy under the Air Commerce Act 1926-1938*. U.S. Department of Transportation, Federal Aviation Administration (1978).

[31] Latour, B. *Science in Action: How to Follow Scientists and Engineers Through Society*. Harvard University Press (1987). ISBN 9780674792913.
`https://books.google.com/books?id=sC4bk4DZXTQC`

[32] Latour, Bruno. *Reassembling the Social: An Introduction to Actor-Network-Theory: An Introduction to Actor-Network-Theory*. Oxford University Press (July 2005).

[33] Law, J. and Lodge, P. *Science for Social Scientists*. Palgrave Macmillan UK (1984). ISBN 9781349175369.
`https://books.google.com/books?id=0eOwCwAAQBAJ`

[34] Meadows, Catherine and Pavlovic, Dusko. *Formalizing Physical Security Procedures*. *Lecture Notes in Computer Science*, 7783 (2013).

[35] Microsoft. *Secured Boot and Measured Boot: Hardening Early Boot Components against Malware*. Technical report, Microsoft (2012).

[36] of Aircraft Accidents Archives, Bureau. *Crash statistics*.
`http://www.baaa-acro.com/general-statistics/crashs-statistics/`

[37] of Fire Underwriters, New York Board. *A Standard for Electric Light Eires, Lamps, etc.* 1. Mike Holt Enterprises of Leesburg, Inc. (1999).

[38] of Transportation, US Dept. *Annual - 1981 to 1995*.
`https://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/subject_areas/airline_information/air_carrier_traffic_statistics/airtraffic/annual/1981_present.html`

[39] of Transportation, US Dept. *Historical Air Traffic Statistics, Annual 1954-1980*.
`https://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/subject_areas/airline_information/air_carrier_traffic_statistics/airtraffic/annual/1954_1980.html`

[40] Pavlovic, Dusko and Meadows, Catherine. *Actor-network procedures: Modeling multi-factor authentication, device pairing, social interactions*. *CoRR*, abs/1106.0706 (2011).
`http://arxiv.org/abs/1106.0706`

[41] Ranter, Harro. *Preliminary ASN data show 2016 to be one of the safest years in aviation history* (Dec 2016).
`https://news.aviation-safety.net/2016/12/29/preliminary-asn-data-show-2016-to-be-one-of-the-safest-years-in-aviation-history/`

[42] Regenscheid, Andrew and Scarfone, Karen. *BIOS Integrity Measurement Guidelines*. 800-155 (Draft). NIST (December 2011).

[43] Rosenblatt, Roger. *A New World Dawns* (1983).

[44] Sime, Jonathan D. *Safety in the Built Environment*. E. & F.N. Spon, London and New York (1988).

[45] Sinofsky, Steven. *Protecting the pre-OS environment with UEFI* (2011). Https://blogs.msdn.microsoft.com/b8/2011/09/22/protecting-the-pre-os-environment-with-uefi/.

[46] TCG. *TCG PC Client Specific Implementation Specification for Conventional BIOS*, 1.21 edition (February 2012).

[47] TCG. *TPM Library Specification*, 2.0 edition (October 2014).

[48] TCG. *TCG PC Client Platform Physical Presence Interface Specification*, 1.30 edition (July 2015).

[49] UEFI. *UEFI Specification Version 2.5* (2015).

[50] Wiens, Jordan. *A Tipping Point For The Trusted Platform Module?* (2008). Http://www.darkreading.com/risk-management/a-tipping-point-for-the-trusted-platform-module/d/d-id/1069284?

[51] Wilkins, Richard and Richardson, Brian. *UEFI Secure Boot in Modern Computer Security Solutions. UEFI Forum* (2013).

[52] Wojtczuk, Rafal and Kallenberg, Corey. *Attacking UEFI Boot Script*. In *31st Chaos Communication Congress* (2014).

[53] Yamamura, Motoaki. *W95.CIH*. http://www.symantec.com/security_response/ writeup.jsp?docid=2000-122010-2655-99 (2002).

[54] Zhang, Zhangkai et al. *Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping* (2017).

[55] Zhou, Zhen-Liu et al. *Research and Implementation of Trusted BIOS Based on UEFI. Computer Engineering*, 34(8):174 (2008).

[56] Zimmer, Vincent J and Rothman, Michael A. *System and method to secure boot UEFI firmware and UEFI-aware operating systems on a mobile internet device (mid)* (2008). US Patent App. 12/165,593.