# CHAINING LAYERED INTEGRITY CHECKS

William A. Arbaugh

A Dissertation

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial

Fulfillment of the Requirements for the Degree of Doctor of Philosophy

1999

Supervisor of Dissertation

Jean Gallier

Graduate Group Chairperson

# Acknowledgements

No work the size of this dissertation is done in isolation, and I would like to thank the people who worked with and supported me over the last four years.

Harold F. Bower has worked with me on numerous occasions. He found and added the entry points in the BIOS source to call AEGIS. He also served as a sounding board for me in the design of AEGIS, and the AEGIS interrupt service routine (ISR). Hal and I also worked together on a pre-cursor of AEGIS, the Security Enhanced Processor (SEP). The problems encountered with the SEP project lead to AEGIS. Hal is also responsible for RATBAG which is described in Chapter 3.

Angelos Keromytis and I jointly designed the protocol used with the AEGIS network recovery and DHCP++. Angelos also served as the ideal person to discuss ideas. He is never shy about telling someone that their idea is nuts.

Scott Alexander, Angelos, and I worked together on the design of SANE, Section 7.1. Scott's contributions are "above the OS", and mine are "below the OS". Angelos worked with both Scott and myself, and developed the naming and threat models.

Ralph Droms et. al. developed the DHCP authentication scheme described in Section 7.2. I developed the delayed aspect of the authentication mechanism along with the threat model.

Chuck Davin worked with me on SIBL, Section 7.3. Chuck designed the network level aspects of SIBL, and I designed from the operating system down.

None of this would have been possible without my advisor, Jonathan Smith. Jonathan allowed me to pursue AEGIS and helped me see the big picture. He both pushed and praised me when needed. Above all, he always treated me as a peer- not an indentured student.

Finally, I would like to thank my wife and children (Lisa, Jon, and Sarah). They are the ones who suffered through my late nights and long weekends working. Without their support and understanding, I would not have finished.

# ABSTRACT

## Chaining Layered Integrity Checks

William A. Arbaugh

Jonathan M. Smith

“Доверяы, но Проверяы”

*“Trust, but Verify”*[1]

In a system, the integrity of lower layers is typically treated as axiomatic by higher layers. Under the presumption that the hardware comprising the system (the lowest layer) is valid, the integrity of a layer can be guaranteed *if and only if:* (1) the integrity of the lower layers is checked, and (2) transitions to higher layers occur only after integrity checks on them are complete. The resulting integrity “chain” inductively guarantees system integrity.

If the integrity chain is not preserved between one or more layers in a system design, the integrity of the system as a whole cannot be guaranteed. In developing the Chaining Layered Integrity Check (CLIC) model, my examination of existing computer systems showed that their architecture overwhelmingly failed to provide integrity guarantees. The architectures I studied either failed to start in a verifiable state or failed to check the integrity of higher levels before a control transition. To address this problem, I have designed a composition model that provides integrity guarantees entitled “Chaining Layered Integrity Checks (CLIC)”. CLIC provides, for the first time, a realistic composition model which system designers can use to aid in the design and

[1]Old Russian Saying used by many over the years.

implementation of *real* systems that require strong integrity guarantees.

To demonstrate the value of CLIC, I designed and implemented a secure and reliable "bootstrap" of a computer system entitled AEGIS. AEGIS is the first *complete* implementation of a secure bootstrap.

As a result of my research, high integrity systems can now be designed and implemented using a well defined and minimal set of trust assumptions using CLIC. This permits integrity guarantees to propagate from the lowest layers of a computer system architecture to the highest layers providing for the first time strong integrity guarantees for a diverse set of applications such as Internet commerce, security systems, and "Active Networks."

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Systems are organized as layers to limit complexity. A common layering principle is the use of levels of abstraction to mark layer boundaries. A computer system is organized in a series of abstraction levels, each of which defines a "virtual machine" upon which higher levels of abstraction are constructed. Each of the virtual machines presupposes that it is operating in an environment where the abstractions of underlying layers can be treated as axiomatic. When these suppositions are true, the system is said to possess *integrity*. When these suppositions are not true, the system may not operate as expected, *i.e.* the system is *untrusted*.

Thus, any system is only as secure as the foundation upon which it is built. For example, a number of attempts were made in the 1960s and 1970s to produce secure computing systems using a secure operating system environment as a basis [Sch75]. An essential presumption of the security arguments for these designs was that the system layers underpinning the operating system, whether hardware, firmware, or both, were trusted. We find it surprising, given the great attention paid to operating system security [MBD89] [EKO94] over the years that so little attention has been paid to the underpinnings required for secure operation, *e.g.,* a secure bootstrapping phase for these operating systems. In effect, these "secure" systems were operating in an environment that was established by "unsecure" software.

Over the years Computer Security research has focused primarily on the prevention of unauthorized disclosure of information, *i.e.* confidentiality of data, rather than the integrity of the system and its data. This focus is due, in part, to the Department of Defense (DOD) Trusted Computer System Evaluation Criteria (TCSEC)[DOD85]. The TCSEC concerned itself primarily with *access to information* rather than the integrity of the system and its information. As a result, integrity measures, which should be fundamental in a secure system, are not. The need for fundamental system integrity is clearly seen by examining the actions of computer crackers'. Regardless of their method of gaining access to the system, the crackers first step usually is to modify the system, violating the systems integrity, to provide continued access [Pen98a, Pen98b]. The success of TRIPWIRE [KS93], an add-on system integrity package[1], which has detected numerous system break-ins [KS94] demonstrates the potential gain from increased integrity protection. Violation of confidentiality, typically, comes after an integrity violation. It should be noted that there can be cases where the above is not true, and confidentiality *must* remain a focus of security research. However, concern for the integrity of the system should be on a par with the concerns for the confidentiality of the data because a confidentiality violation will surely follow an intentional integrity violation.

Modularity and layering were introduced in the 1970's to assist in abstracting difficult problems. During this time, David Parnas presented a layering model to assist in the composition of complex systems [Par74]. As part of the model, Parnas defined a hierarchical structure, and introduced and defined a new relation "USES" [Par77]. While Parnas correctly identified the problem

[1]While most security professionals scoff at "add-on" security systems, TRIPWIRE has been extremely successful despite the fact that it is "add-on" security.

of a higher layer depending on the correctness of lower layers, he did not develop a method for ensuring the correctness of the layers, nor did he present a course of action when the correctness of a layer is identified as invalid.

To address these problems, I have developed the Chaining Layered Integrity Checks (CLIC) composition model. CLIC extends the layering model developed by Parnas to ensure that the integrity of a higher layer remains valid before transferring control to it. Further if the higher layer's integrity is found to be invalid, corrective action can be taken which will provide a limited degree of fault tolerance. As a result, a properly implemented system using CLIC can guarantee the correctness of a system, assuming that the correctness of the system was shown previously and its integrity recorded, by ensuring the integrity remains unchanged.

To demonstrate CLIC's potential and practicality, I have designed and implemented AEGIS, a secure and reliable bootstrap of an IBM PC.

Integrity measures have been used in the IBM PC community for a number of years to detect boot sector viruses. These measures initially started with software packages running as an application of the operating system, and now most BIOS vendors offer the ability to trap writes to the boot sector, assuming that the write is vectored through the BIOS[2]. These steps have effectively eliminated the spread and introduction of most boot sector viruses[3]. As the vendors closed the vector for spreading a boot sector virus, the virus authors advanced to more fruitful territory such as macro viruses. However, the advent of both the Plug-and-Play (PnP) specification [CLC94] and flash chips for storage of the PnP Extended System Configuration Data (ESCD)[Any95] opened a new avenue for virus writers because programming of the flash chip, which stores the BIOS, must be

[2]This is not the case in protected mode operating systems such as WindowsNT and UNIX which rely on device drivers rather than the BIOS.

[3]While boot sector viruses and Trojan horses have primarily been a problem for MS-DOS and Windows based systems there is new evidence that crackers have used the boot block of a UNIX system to hide a back-door[Kla97]

enabled for updates of the ESCD[4]. Prior to July 1998, a flash virus had not been seen in the "wild". One was, however, designed in a laboratory [Sku96][Bon97]. Then in July of 1998, the CIH virus was identified by PC users in Taiwan [RL98]. CIH later spread throughout the world [Fit98]. The CIH virus triggers on the 26th of the month. The payload, contained in a Windows 95 portable executable file, overwrites the BIOS flash ROM with garbage. AEGIS can detect the modifications to the BIOS, and recover a correct copy. The only difference to the user would be that the bootstrap would take slightly longer than normal because of the reboot required after corrective action is taken.

While AEGIS serves as a strong foundation for future security measures, it also can reduce the Total Cost of Ownership (TCO) of IBM personal computers. Automatically detecting and repairing integrity failures permits the user to continue to work without the nuisance of a trouble call to the support staff and the associated down time spent waiting for the repair. A system administrator can monitor the log of the AEGIS trusted repository and identify those workstations that require "hands on" repairs, *e.g.* ROM failure, and schedule the work to be done when the user is not using the computer. In addition to saving the user's time, this permits the system administrator to schedule a trouble call rather than react to it. This greatly reduces the stress on both the user and the administrators.

The research described in this dissertation was started as a result of research I performed at the U.S. Department of Defense in 1993 and 1994. In that work, I modified a SunOS 4.1.3 kernel to execute a binary only if it contained a valid digital signature of the executable [Arb94]. This work was then integrated into the virtual file system of the FreeBSD operating system at the University of Pennsylvania in the fall of 1994. Changes included caching and the verification of interpreted

[4] Assuming that the BIOS uses Flash for ESCD storage. It is not required to use Flash, but some BIOS implementations do.

programs, *e.g.* perl and shell programs [Arb95]. While analyzing these prototypes, it was clear the most effective way to circumvent the security measures in the operating system was to modify the kernel. Continuing this logic, resulted in the observation that a "chain of integrity" must be established starting as low as possible.

## 1.1 Novelty

This research is novel in two respects. First, this research provides the first model for composing system layers to provide strong integrity guarantees for an entire system, and provided that the layers are "trustworthy", the first model for composing a "trustworthy" system. Second, AEGIS is the first system to propose and implement the *automatic recovery* of corrupt or invalid bootstrap components, providing a significant increase in reliability and mitigation of denial of service attacks. Associated with this is the ability to *centrally manage* the bootstrap process from a remote location, greatly reducing the TCO of an enterprise.

While others (see Chapter 3) have proposed, and in some cases implemented, a partial "secure" bootstrap, the work has either been flawed or required significant architecture changes to commodity hardware, and thus has not been completely implemented. Furthermore, AEGIS is the first system to propose and implement a secure bootstrap process that is *practical* in that either no, or a very limited number of, changes are needed when using commercial off the shelf equipment. Finally, AEGIS is the first to consider and mitigate certain classes of denial of service attacks against the bootstrap process.

## 1.2 What's Not Covered in this Dissertation

This dissertation does not cover several important topics beyond the scope of this work. The first is *trust management*. Trust management handles the delicate task of determining the proper delegation of authority between principles. The Keynote system is making good progress on this difficult problem [BFIK98a]. A second area not covered is the verification and validation of programs as "trustworthy" therefore ensuring the integrity of a "buggy" or "malicious" program- preserves the bugs or malice. Fortunately, this area has several novel efforts underway [NL96a] [BK95] as well.

## 1.3 Remainder of this Dissertation

The next chapter presents a short introduction to several technical areas that will be helpful in understanding the remainder of the dissertation. Chapter 3 presents related work. Chapter 4 describes the Chaining Layered Integrity Checks Model and a formal proof that CLIC provides *strong* and *reliable* integrity guarantees. Chapter 5 presents the architecture of an embodiment of CLIC, the AEGIS secure bootstrap, a secure and reliable bootstrap process for the Intel x86 architecture. This is followed by Chapter 6 which presents the specific details of the implementation of AEGIS and the experimental results obtained from testing. Chapter 7 presents several applications based on or derived from AEGIS and possible future work. Finally, Chapter 8 presents the conclusions drawn from this research.

# Chapter 2

# Technical Foundations

This section provides a short introduction to several important subjects. The material presented will help in understanding the remainder of the dissertation.

The first section presents several definitions used throughout the remainder of this dissertation. The second section describes the process that an IBM personal computer uses during start-up. The next five sections cover aspects of public key cryptography, digital signatures, and Diffie-Hellman key exchange. The eighth and the ninth sections cover the Dynamic Host Configuration Protocol (DHCP), and the Trivial File Transfer Protocol (TFTP). The chapter concludes by describing the various types of ROM available to a system designer.

## 2.1 Definitions

The term *trust* is used heavily in Computer Security. Unfortunately, the term has several definitions depending on who uses it and how the term is used. In fact, the U.S. Department of Defense's *Orange Book*, which defined several levels of security a computer host could provide, defines *trust* ambiguously. Throughout this dissertation, I use *trust* definitions which are slight modifications of Peter Neumann's [Neu95]. I define an object as *trusted* when the object always operates as expected according to design and policy. A stronger trust statement is when an object is *trustworthy*. A *trustworthy* object is one that has been shown in some convincing manner, *e.g.* a formal code-review or formal mathematical analysis, to operate as expected. A *security-critical* object is one for which the security (defined by a policy) of the system depends on the proper operation of the object. A security-critical object can be considered trusted, which is usually the case in most secure systems, but unfortunately this assumption leads to an unnecessary profusion of such objects.

We note the distinction between trust and integrity: trustworthiness is determined through the verification of objects and the dependencies among them. Integrity demonstrates that objects haven't been modified. Thus integrity checking in a trustworthy system is about preserving a trust or trust relationship that was previously established.

## 2.2 IBM Personal Computer Boot Process

Every computer with the IBM PC architecture follows approximately the same boot process. I have divided this process into four levels of abstraction (see figure 2.1), which correspond to phases of the bootstrap operation. The first phase is the Power on Self Test or POST [Ltd91]. POST is invoked in one of four ways:

1. Applying power to the computer automatically invokes POST, causing the processor to jump to the entry point indicated by the processor reset vector located at address FFFF:0000h.

2. Hardware reset also causes the processor to jump to the entry point indicated by the processor reset vector.

3. Warm boot (*ctrl-alt-del* under DOS) invokes POST without testing or initializing the upper

64K of system memory.

4. Software programs, if permitted by the operating system, can jump to the processor reset vector.

In each of the cases above, a sequence of tests are conducted. All of these tests, except for the initial processor self test, are under the control of the system BIOS. The POST process first tests and then initializes the hardware devices found on the motherboard. The system configuration data stored on the motherboard are then tested and initialized/loaded. Finally, the hardware devices that exist on add-in boards are tested and initialized. The hardware devices are tested in the approximate order shown below:

1. Microprocessor,
2. CMOS RAM,
3. Checksum of BIOS ROM,
4. System Chip sets,
5. First Programmable timer counter,
6. DMA controller and page registers,
7. Base 64K of system RAM,
8. Serial and parallel ports,
9. Programmable Interrupt controller,
10. Keyboard controller,

11. CMOS data,
12. Video controller,
13. Real time clock,
14. Microprocessor shutdown,
15. Programmable timer counter 2,
16. Keyboard,
17. Mouse,
18. System RAM above 64K,
19. Fixed disk and diskette controllers,
20. Shadow RAM areas,
21. Option ROMs,
22. Cache controller,
23. Cache.

Once the BIOS has performed all of its power on tests, it begins searching for expansion card ROMs which are identified in memory by a specific two byte signature, 0x55AA. The memory area 0xC0000 - 0xC7FFF is searched for video expansion ROMs, and the area 0xC8000 - 0xDFFFF is searched for general purpose ROMs. Each memory area is searched in 2KB increments. Finally, 0xE0000 is checked for a 64KB system expansion ROM. Once a ROM signature is found by the

| Level | Description |
|---|---|
| 0 | Processor self test |
| 1 | BIOS initialization and component initialization checks |
| 2 | Expansion ROM search and initialization |
| 3 | Boot block initialization |
| 4 | Operating System initialization |

Table 2.1: Overview of POST process

BIOS, control is passed to it after validating an eight bit additive checksum. When the ROM completes its execution, control is returned to the BIOS.

The final step of the POST process calls the BIOS operating system bootstrap interrupt (Int 0x19). The bootstrap code first finds a bootable disk by searching the disk search order defined in the CMOS. Once it finds a bootable disk, it loads the primary boot block into memory and passes control to it. The code contained in the boot block proceeds to load the operating system, or a secondary boot block depending on the operating system [Gri93] [Eli96], see Figure 2.1

Ideally, the boot process would proceed in a series of levels with each level passing control to the next until the operating system kernel is running. Unfortunately, the IBM architecture uses a "star like" model which is shown in Figure 2.1.

## 2.3 Public Key Cryptography

Public key cryptographic algorithms differ from the traditional symmetric encryption algorithms such as the Data Encryption Standard (DES) [NBS77]. In symmetric systems such as DES, the same key is used to encrypt and decrypt. Public key systems are asymmetric. They use two different keys, a private key, $K_{private}$, and a public key, $K_{public}$ where $D_{K_{public}}(E_{K_{private}}(M)) = M$.

Public key cryptographic systems have a significant advantage over the traditional symmetric

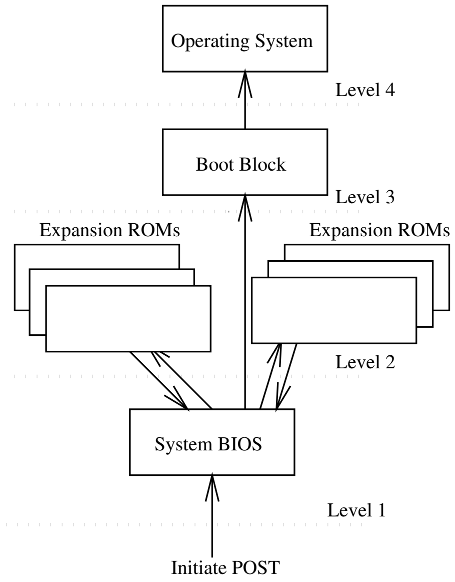


Figure 2.1: IBM PC boot process

systems in that two principals can exchange a message, or verify the validity of a digital object, (see Section 2.5), provided they have access to the originator's public key in some trusted manner. Here, confidentiality of the public key is not required. Whereas in a symmetric system, the principals must exchange keys in a confidential manner, usually out-of-band.

The principals using a public key system will typically publish their public keys in a fashion so that the principal's identity is bound to the public key (see Section 2.6). The mechanism which supports publishing the public key is referred to as a *Public Key Infrastructure* and is a required element to support trustworthy use of a public key system on a wide scale.

## 2.4 Cryptographic Hash Functions

A cryptographic hash function, $H$, is a one-way function with the additional properties that [Sch96]:

**Property 2.1** *Given a message, $M$, it is easy to compute the hash value, $h$.*

**Property 2.2** *Given $h$, it is hard to compute $M$ such that $H(M) = h$.*

**Property 2.3** *Given $M$, it is hard to find another message, $M'$, such that $H(M) = H(M')$.*

## 2.5 Digital Signatures

Digital signatures use a cryptographic hash function and a public key cryptographic system to provide non-repudiation. A principal wishing to sign an object, $A$, will compute a cryptographic digest of $A$, $H(A)$, and encrypt the resulting digest with their private key: $S = E_{K_{private}}(H(A))$. Anyone wishing to validate the signature, $S$, needs only to recompute the cryptographic hash of $A$ and compare it to the value obtained by decrypting $S$: $H(A) == D_{K_{public}}(S)$. If the computed hash matches the one obtained by decrypting $S$, then the signature is validated. Otherwise, the signature is not validated.

## 2.6 Public Key Certificates

A public key certificate binds an identity to a public key. In the case of the X.509 standard [Com89], an identity is represented as a "distinguished name", *e.g.* /C=US/ST=PA/L=Philadelphia/O=University of Pennsylvania/OU=Department of Computer and Information Science/CN=William A. Arbaugh (see Figure 2.2). Whereas in the SPKI/SDSI proposed certificate standard [EFRT97], the identity is represented by a public key or its message digest (see Figure 2.3). Unfortunately, the advantages and disadvantages of each approach are beyond the scope of this introduction.

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 830525701 (0x3180d105)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=Ca, L=Nepean, OU=No Liability Accepted, O=For Demo Purposes
        Validity
            Not Before: Apr 26 13:35:01 1996 GMT
            Not After : Apr 26 13:35:01 2006 GMT
        Subject: C=Ca, L=Nepean, OU=No Liability Accepted, O=For Demo Purpose
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:9a:ae:84:bb:3b:50:c0:d2:19:b8:21:e7:cd:61:
                    4d:5c:c7:f9:ea:3a:6c:c4:9e:4d:bd:d5:dd:57:ab:
                    39:c3:78:5e:30:12:06:a4:6e:40:9c:cd:32:b1:2b:
                    bf:75:c8:da:57:1b:94:66:9e:0a:87:ed:cc:a0:0d:
                    14:c7:64:9e:a6:7c:24:67:d3:01:82:f8:02:34:d0:
                    5d:af:61:31:aa:3c:64:50:2f:53:77:95:94:f4:60:
                    32:85:0b:44:90:78:c3:23:14:86:75:69:e4:22:82:
                    ba:05:42:71:e8:d1:50:77:cd:62:0c:a4:d9:24:6b:
                    c3:d8:f6:5c:47:64:09:96:c1
                Exponent: 3 (0x3)
    Signature Algorithm: md5WithRSAEncryption
        40:c7:45:0c:54:0d:6c:e7:89:4c:43:25:e3:31:7e:64:8f:df:
        45:24:de:62:b5:bf:1b:2b:54:64:f9:c7:a0:3d:03:c5:71:63:
        bd:48:45:b2:12:38:c1:8c:89:23:8f:30:30:06:46:84:b3:31:
        6c:34:6c:5e:9b:1b:57:bf:08:c8:9b:5c:44:50:c4:d5:94:f1:
        16:4e:cd:aa:9c:3b:c0:50:0f:56:62:a8:56:6e:11:f4:b6:81:
        93:df:ad:b7:e9:00:2c:a8:26:7b:ea:b6:d3:45:54:92:5f:60:
        47:c8:9c:09:30:6d:ad:09:1b:fb:91:12:7f:fc:d2:20:c6:3d:
        c7:e3
```

Figure 2.2: Example X509 Certificate

```
((cert (issuer (hash-of-key (hash sha1
                clientkey)))
    (subject (hash-of-key (hash sha1
                  clientkey)))
     (tag (client (cnonce cbytes)
            (msg-hash
             (hash sha1 hbytes))))
     (not-before 09/01/97-0000)
     (not-after 09/01/97-0000))
 (signature (hash sha1 hashbytes)
  (public-key dsa-sha1 clientkey)
  (sigbytes)))
```

Figure 2.3: Example SDSI Certificate

## 2.7 Diffie-Hellman Key Exchange

The Diffie Hellman Key Agreement (DH) [DH76] permits two parties to establish a shared secret. Unfortunately, the algorithm as originally proposed is susceptible to a man-in-the-middle attack. The attack can be defeated, however, by combining DH with a public key algorithm such as DSA or RSA as proposed in the Station to Station Protocol (StS)[DvOW92]. The AEGIS recovery protocol is an extension to StS.

The algorithm is based on the difficulty of calculating discrete logarithms in a finite field. Each participant agrees to two primes, $g$ and $p$, such that $g$ is primitive $mod\, p$. These values do not need to be protected in order to ensure the strength of the system, and therefore can be public values. Each participant then generates a secret large random integer. Bob generates $x$ as his large random integer and computes $X = g^x \mod p$. He then sends $X$ to Alice. Alice generates $y$ as her large random integer and computes $Y = g^y \mod p$. She then sends $Y$ to Bob. Bob and Alice can now each compute a shared secret, $k$, by computing $k = Y^x \mod p$ and $k = X^y \mod p$, respectively. The values $X$, $Y$, $g$, and $p$ can all be made public without loss of security.

## 2.8 Dynamic Host Configuration Protocol

The DHCP protocol[Dro97] provides clients the ability to configure their networking and host specific parameters dynamically during the boot process. The use of DHCP is increasing dramatically for two main reasons. The first is an effort to ease the cost of network management. The second is a proliferation of network appliances such as printers and toasters. The typical parameters are the IP addresses of the client, gateways, and DNS server. DHCP, however, supports up to 255 configuration parameters, or options. Currently, approximately one hundred options are defined for DHCP [AD97]. One of these options is an authentication option, described in Section 7.2.

The format of a DHCP message is shown in Figure 2.4[Dro97]. The first field in the DHCP

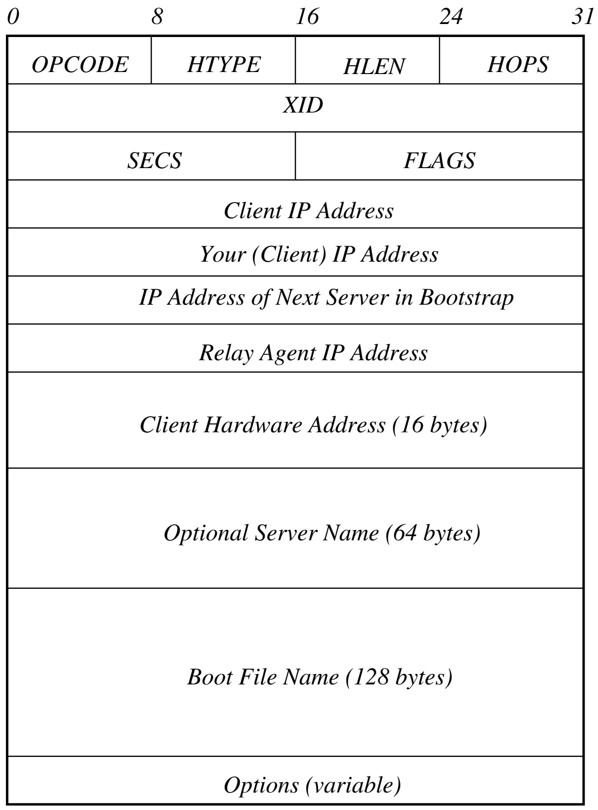


Figure 2.4: DHCP Message Format

message is the *opcode*. The opcode can have one of two values, 1 for a BOOTREQUEST message, and 2 for a BOOTREPLY message. The next field, *htype*, is the hardware address type defined by the "Assigned Numbers" RFC[RP94]. *hlen* indicates the length of the hardware address. *hops* is set to zero by the client and used by BOOTP relay agents to determine if they should forward the message. *xid* is a random number chosen by the client to permit the client and the server to associate messages between each other. *secs* is set by the client to the number of seconds elapsed since the start address acquisition process. Currently, only the leftmost bit of the *flags* field is used to help solve an IP multicast problem. The remaining bits must be zero. *ciaddr* is the client address if the client knows it already, *yiaddr* is "your" address set by the server if the client did not know its address (or had a bad one). *giaddr* is the relay agent address. *chaddr* is the client's hardware address. *sname* is an optional null terminated string containing the server's name. *file* is the name of the boot file. In AEGIS, this is the name of the component to recover. Finally, *options* is a variable length field containing any options associated with the message.

The initial message exchange between the client and the server is shown in Figure 2.5.

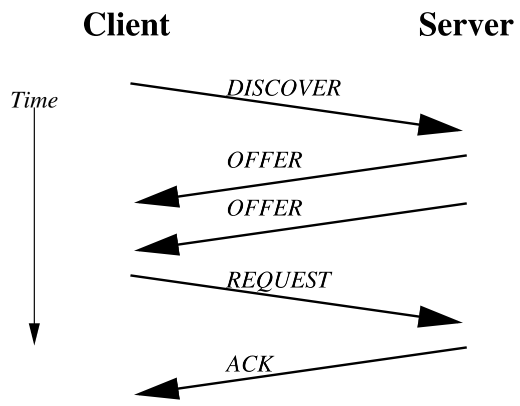


Figure 2.5: Initial DHCP Message Exchange

The client begins the process by sending a DHCPDISCOVER message as a broadcast message on its local area network. The broadcast message may or may not be forwarded beyond the LAN depending on the existence of relay agents at the gateways. Any or all DHCP servers respond with a DHCPOFFER message. The client selects one of the DHCPOFFER messages and responds to that server with a DHCPREQUEST message, and the server acknowledges it with a DHCPACK.

In addition to providing networking and host specific parameters, DHCP can provide the name and server location of a bootstrap program to support diskless clients. After the client receives the IP address of the boot server and the name of the bootstrap program, the client uses TFTP[Sol92] to contact the server and transfer the file. At present the DHCP standard, as defined by the Internet Engineering Task Force, does not include *any* authentication mechanism.

## 2.9 Trivial File Transfer Protocol

TFTP was designed to be simple and small enough to fit into a ROM on a diskless client. Because of this, TFTP uses UDP rather than TCP with no authentication included in the protocol. TFTP has five unique messages that are identified by a two byte opcode value at the beginning of the packet. The Read Request (RRQ) and the Write Request (WRQ) packets share the same format and have an options capability[MH95]. Unfortunately, the option capability does not apply to the remaining three packet types (DATA, ACK, and ERROR). This makes it problematic to use a MAC with TFTP without changing the protocol itself.

## 2.10 Flash ROM

Flash ROM was introduced a number of years ago as a lower cost alternative to EEPROM. The biggest difference between Flash ROM and EEPROM is that EEPROM can be reprogrammed/erased at the byte level and a Flash ROM must be reprogrammed/erased on the block level.

Intel now provides for three different data blocks, boot block, parameter block, and main block, on its latest generation of flash ROM[Haz95a]. Each block type emulates a different type of firmware storage device. The boot block emulates a ROM by using additional hardware protection (setting pin RP# high) to prevent accidental or malicious erasure of that block of the flash chip. The parameter block emulates EEPROM or battery-backed SRAM by permitting programming of the individual blocks rather than the entire chip. The final block type is called a main block. This block stores the system firmware as would a normal flash chip. The memory map for an Intel 28F001BX-T, the chip being used with AEGIS for the BIOS, is shown in Figure 2.6. The idea behind the boot block is to provide users with the ability to boot the computer in a limited fashion to permit re-flashing of the BIOS with a floppy in the event it becomes overwritten with garbage or buggy code [Haz95b]. This eliminates the need for the user to pull the chip from the board and to find a ROM programmer when there is a catastrophic failure[1]. The idea behind the parameter blocks is to permit "Plug and Play" data storage on the chip without the cost of reprogramming the entire chip [Any95]. The fact that "Plug and Play" requires non-volatile storage, however, is one of the main reasons that portions of the flash ROM must remain writable and thus vulnerable to flash BIOS viruses.

[1]This also permits recovery from viruses such as the CIH virus

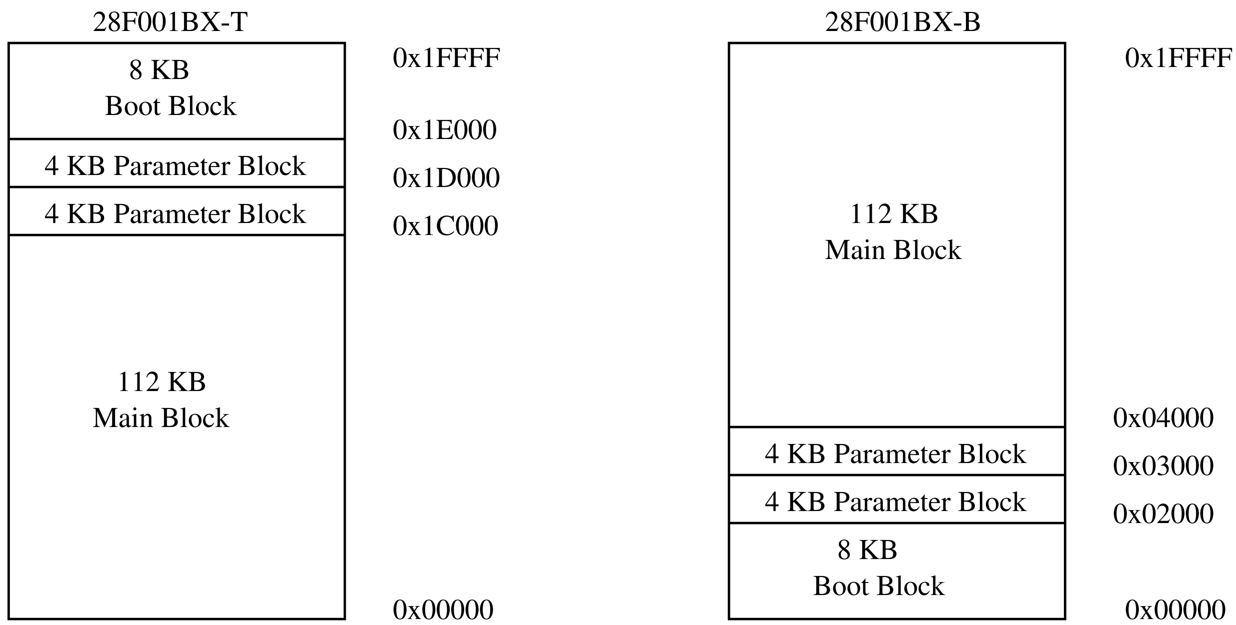


Figure 2.6: Intel 28F001BX-T and 28F001BX-B 1-MBit Flash ROM Memory Maps

# Chapter 3

# Previous Work

While computer science is still a young science when compared to the classical sciences, almost all computer science research today leverages off of previous work in some manner. This section presents several research areas that this dissertation has either leveraged or reviewed.

## 3.1 Secure Bootstrap

When compared to the sizable amount of prior research on trusted operating systems, relatively little prior work has been done on a secure bootstrap. There are many reasons why this is so. The first and primary reason is DOD's focus on confidentiality instead of integrity. The second is that most assume that physical security measures can sufficiently protect the bootstrap process. The third and final reason is that designing and engineering a valid model is a difficult process. The remainder of this section presents past efforts at securing the bootstrap process.

### 3.1.1 Lampson

Lampson [LAB92] describes a secure boot model as an example for his authentication calculus. In Lampson's model, the entire boot ROM is trusted, and he does not address the verification of expansion cards/ROMs.

### 3.1.2 Birlix

The Birlix [HKK93] Security Architecture proposes a model designed by Michael Gross that is similar to Lampson's. In Birlix, a system boots in five steps:

1. The operating system code is loaded.

2. The boot ROM verifies a digital signature of the operating system.

3. A new private and public key are generated by the boot process for use until the next boot. It is the responsibility of the operating system once it is started to protect this private key

4. A boot certificate binding the identity of the node with the newly generated public key and several other attributes is generated. This certificate is signed by a private key belonging to the node's processor. The certificate serves two purposes. The first is to demonstrate that the node completed a valid secure boot process, and the second is for future authentication with other nodes.

5. Finally, the operating system is started.

The Birlix model, unfortunately, does not validate the integrity of any of the lower level firmware, expansion ROMs or boot blocks. As a result, it leaves itself open to exactly the types of problems Birlix was attempting to prevent.

### 3.1.3 BITS

Clark [Cla94][CH94] presents a secure boot process (BITS) for DOS that utilizes an 8-bit expansion board, a smart card drive mounted in a floppy bay, software for configuration, software for file verification, and finally a smart card. BITS stores some of the OS bootstrap code on the smart card as well as checksums for the remaining bootstrap components. BITS takes control of the bootstrap process when the expansion board ROM containing the BITS code is found and control passed to it by the BIOS. The BITS code in the expansion ROM initializes the smart card reader and permits the user to authenticate with the smartcard by entering the correct PIN or pass phrase, thus "unlocking" the smart card. The host then authenticates the card by determining if it holds a shared secret established previously between the host (or group of hosts) and the smart card. If both authentications complete correctly, the host uploads the OS boot code from the smart card and uses any stored checksums to check the integrity of the remaining OS components.

Clark's approach is novel in that it provides a strong (under the assumption of the physical security of the host) authentication mechanism between the user and the host through the use of the smart card. BITS, however, fails to provide a process for a secure bootstrap since BITS takes control of the process during level 2 (Expansion ROM search). As a result, BITS only ensures the integrity of the operating system components and ignores the firmware (BIOS and expansion ROMs). BITS also suffers from maintenance issues involving the storage of bootstrap components on the smart card and the use of checksums instead of public key cryptography. As a result, the user *must* have a separate smart card for every host that they use, and any time the OS changes on a host the associated smart cards must be updated as well. This results in significant overhead for administrators.

### 3.1.4 Yee

The first public presentation of a secure boot process was done by Yee [TY91]. In Yee's model, a cryptographic co-processor is the first to gain control of the system. Unfortunately, this is not easily done without a complete architectural revision of most computer systems or the addition of an expensive daughter board as used in hardware in-circuit emulators. Yee expands his discussion of a secure boot in his thesis [Yee94], but he continues to state that the secure coprocessor should control the boot process by verifying each component prior to its use. Yee also states that boot ROM modifications *may* be required, but since a complete prototype secure boot process was never implemented a number of implementation questions remain unanswered concerning the ROM modifications.

Yee integrated a prototype of the IBM 4758 PCI Cryptographic Co-processor into the microchannel bus of an IBM PS/2. Unfortunately, the prototype card from IBM could not bus master and access memory directly. This coupled with other problems prevented Yee from completing a secure boot prototype. Instead, Yee focused on the important integration issues involved with a secure co-processor.

### 3.1.5 Arnold and Jablon Patents

Two patents were recently found during a prior art search. The first patent is, #5,379,342 *Method and Apparatus for Providing Enhanced Data Verification in a Computer System*, by William C. Arnold and assigned to International Business Machines Corp. The second patent is #5,421,006 *Method and Apparatus for Assessing Integrity of Computer System Software* by David P. Jablon and assigned to Compaq Computer Corp. Both of these patents are similar in that the BIOS verifies the boot block before control is transfered and the boot block verifies the OS kernel before control is transferred (see Figure 3.1). But Jablon continues to provide static integrity checks while the operating system is running, i.e prior to execution of a program, as in Pozzo and Grey[PG89]. Another difference between the two patents is that Arnold uses a Modification Detection Code, *e.g.* MD5, and Jablon uses public key cryptography. Both approaches, however, fail to verify the

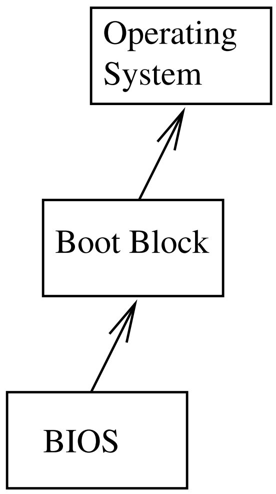


Figure 3.1: Control Flow for Arnold and Jablon Patents

BIOS beyond the normal eight bit additive CRC, making the system vulnerable to a CIH like attack. Both approaches also fail to verify expansion ROMs. This is surprising since Jablon states, "once an untrusted program is run, the entire system is then said to be in an untrusted mode." The ROMs on add-in boards are programs, and they are run during the boot process without verification. Therefore, Jablon's approach fails to provide a secure bootstrap process, as does Arnold's, since neither approach verifies expansion ROMs. Additionally, neither patent included a recovery process beyond the insertion of a floppy disk in the event of failure.

### 3.1.6 RATBAG

RATBAG was an internal U.S. Department of Defense project started in 1990 to review the security of personal computers. A custom BIOS resulted from the two and half year project that provided similar functionality as the Arnold patent [Jen93]. RATBAG is unique in that a complete re-engineering of the BIOS was performed. Harold Bower, who assisted with the AEGIS prototype, lead the RATBAG effort.

### 3.1.7 Anti-Virus Products

A number of companies and BIOS vendors have anti-virus capabilities in their products. All concern themselves with the boot block only. Those products that run as an application over the operating system typically store a modification detection code for the boot block and check it when they are run. This detects changes to the boot block, but is susceptible to spoofing. The BIOS anti-virus protection simply alerts the user when a process is attempting to write to the boot block. The protection is ineffective when a protected mode operating system is running, or a real mode application writes directly to the storage device, *i.e.* without the use of the BIOS services. Finally, several vendors are now offering ROM-based anti-virus protection. These products work by using an expansion ROM board that is executed during the boot process. The code on the ROM board checks the boot block against a previously stored MDC in order to detect changes. The vendors claim this prevents the possibility of spoofing the check. This is not entirely true, since the BIOS passes control to the ROM and can skip passing control to the ROM if reprogrammed to do so completely by-passing the security measure.

### 3.1.8 Summary of Previous Work

Table 3.1 summarizes the previous work by listing those efforts which were implemented, and those that provide a secure design or implementation.

| Principal Investigator | Design Implemented | Secure Design |
|---|---|---|
| Lampson | No | No |
| Birlix | Yes | No |
| BITS | Yes | No |
| Yee | Partial | Yes |
| Arnold | Unknown | No |
| Jablon | Unknown | No |
| Anti Virus Products | Yes | No |

Table 3.1: Summary of Secure Bootstrap Previous Work

## 3.2 Automatic Recovery of Bootstrap Components

Previous efforts to provide recovery of bootstrap components have required human intervention, typically the insertion of a floppy disk containing the new component. Additionally, these efforts have only focused on repairing a single component of the entire process, *i.e.* only repairing the boot block, or the BIOS, but not both. This is in contrast to AEGIS' approach which will provide automatic recovery of *all* of the bootstrap components, including ROM chips.

# Chapter 4

# Chaining Layered Integrity Checks

In a system, the integrity of lower layers is typically treated as axiomatic by higher layers. Under the presumption that the hardware comprising the system (the lowest layer) is valid, the integrity of a layer can be guaranteed *if and only if:* (1) the integrity of the lower layers is checked, and (2) transitions to higher layers occur only after integrity checks on them are complete. The resulting integrity "chain" inductively guarantees system integrity.

If the integrity chain is not preserved between one or more layers in a system design, the integrity of the system as a whole cannot be guaranteed. In developing the Chaining Layered Integrity Check (CLIC) model, my examination of existing computer systems showed that their architecture overwhelmingly failed to provide integrity guarantees. The architectures I studied either failed to start in a verifiable state or failed to check the integrity of higher levels before a control transition. To address this problem, I have designed a composition model that provides integrity guarantees entitled "Chaining Layered Integrity Checks (CLIC)". CLIC provides, for the first time, a realistic composition model which system designers can use to aid in the design and implementation of *real* systems that require strong integrity guarantees.

This chapter formalizes the CLIC model. The first section begins by defining the properties of a layer in CLIC. The second section describes how to compose layers, and the third section describes the failure actions. Finally, the fourth and last section defines the CLIC relation and presents a formal proof that CLIC ensures the integrity of a system composed using CLIC.

## 4.1 Defining a Layer

A layer is the fundamental building block in CLIC, representing an increase in functionality over that already provided. The main goal of a layer is to perform some operation, export an interface to an operation(s), or both. In exporting an interface, the layer provides an abstraction to reduce the complexity of programming the objects below the layer, *e.g.* the hardware or other layers.

A CLIC layer has the following properties:

**Property 4.1 (Static Integrity)** *A layer does not pass execution to another layer without first verifying the integrity of the called layer.*

**Property 4.2 (Dynamic Integrity)** *A layer does not modify the text of any layer- including its own.*

The Static Integrity property ensures that the representation of a layer has not changed from a previously recorded representation. This property prevents the addition of either corrupt or malicious layers into the system. Enforcement of the Static Integrity property is accomplished through the use of the Transition Validation Function (TVF) described below (see Section 4.2.1).

The Dynamic Integrity property ensures that the layers already in the system do not change themselves, *i.e.* they are not self-modifying code, and do not change other layers. Such changes could potentially introduce malice. Enforcement of the Dynamic Integrity property should be accomplished through the use of the memory protection hardware provided by the processor, and through a thorough review of the layer (see the Assumption below), to identify and remove any code violating this property.

The definition of a layer only requires one assumption:

**Assumption 4.1** *A layer must be "trustworthy". This can be determined by a review process such as a code walk-through or formal analysis that ensures the properties discussed above are maintained.*

While this is a significant assumption, it is one that must be made for any piece of security software with or without CLIC.

## 4.2 Composing Layers into Systems

Systems are organized as layers to limit complexity. A common layering principle is the use of levels of abstraction to mark layer boundaries. A computer system is organized in a series of abstraction levels, each of which defines a "virtual machine" upon which higher levels of abstraction are constructed. This section describes how a series of layers can be composed such that the integrity of the resultant system remains valid.

A composed system will have at least one base layer[1], or entry point for the system. A base layer, as a result of its special status within the system as the entry point, requires an additional property beyond those associated with other layers.

**Property 4.3 (Base)** *The lowest layer, or base layer, in addition to meeting the properties presented earlier, must be represented on non-mutable storage, i.e. the layer can not be stored on an*

[1]In this dissertation, I am only concerned with systems with a single base layer.

*EEPROM or Flash ROM without additional protections requiring physical access to reprogram the device.*

The Base property provides a "virtual" ground for integrity. The boot block of an Intel Flash ROM (see section 2.10) with motherboard support can easily provide this property given the following additional assumption.

**Assumption 4.2 (Physical Security)** *The physical security of the hardware and the base layer comprising the system is protected through a comprehensive physical security program.*

The physical security assumption is a necessary requirement for the same reason that CLIC is a requirement. The hardware comprises the lowest possible layer in a system, and its integrity must be valid to ensure the integrity of layers built on top. Fortunately, a number of techniques using both physical and electronic security can be used to assist in ensuring the integrity of the system's hardware. Discussion of these measures, however, is beyond the scope of this dissertation.

The composition of two layers into a system usually creates a dependency relationship between the layers. Otherwise there is no need to compose the layers as the layering results in an orphaned layer. David Parnas in 1974 examined several dependency relationships [Par74, Par77]. One relation studied by Parnas is the *USES* relation, defined below. *USES* captures the control flow of a system composed of subprograms, layers, or modules.

**Relation 1**

$USES(p_j, p_i) = iff$ $p_i$ *calls* $p_j$ *and* $p_i$ *will be considered incorrect if* $p_j$ *does not function properly.*

In CLIC, I use the *USES* relation to induce an ordering of the layers as they are composed with the base layer as $L_0$.

### 4.2.1 Transition Validation Function

The *USES* relation requires one layer to *call* or transfer control to another layer. In CLIC, this is accomplished via the Transition Validation Function (TVF) rather than the usual call or jump instruction. The purpose of the TVF is to ensure the static integrity property of the new layer.

The TVF must first validate the integrity of the called layer before transferring control. This is done by computing a value representing the integrity of the called layer using a cryptographic hash function, $H$, having the properties defined in Section 2.4. The computed value, $h_{t'}$, is then compared to a previously computed value, $h_t$. If the two values are the same, then the TVF returns true. Otherwise, the TVF returns false. The TVF is defined below in Equation 4.1,

$$TVF(L) = (h_t == h_{t'}) \tag{4.1}$$

where $h_t = H(L)$ at time $t$, $h_{t'} = H(L)$ at time $t'$ for $t' > t$.

The previously computed hash value, $h_t$, for each level must also be protected from tampering either by storing the value on non-mutable storage such as a ROM, or by using an asymmetric cryptographic algorithm to produce a digital signature (see Section 2.5).

## 4.3 Failure Actions

When a composed system detects an integrity failure caused by the TVF returning false, one of three possible courses of action can be taken.

The first is to continue normally, but issue a warning. Unfortunately, this may result in the execution or use of either a corrupt or malicious layer.

The second is to not use or execute the layer. This approach is typically called *fail secure*, and creates a potential denial of service attack.

The final approach is to recover and correct the inconsistency from a *trusted repository* before the use or execution of the layer.

The first two approaches are unacceptable when the system is an important network element such as a switche, or an intrusion detection monitor, or it is associated with electronic commerce, since these approaches either make the layer unavailable for service, or its results un-trustworthy. There may be times, however, when a valid layer can not be recovered. In those cases, it is up to policy to determine as to what occurs. The default action is to halt.

The trusted repository can either be a network host, or an expansion ROM board. If the repository is an expansion ROM board, then the ROM board should have both the base and physical security properties. If the repository is a network host, then the recovery of the layer must be performed using a secure protocol with at least the same cryptographic strength as the transition validation function.

## 4.4 CLIC Relation

The CLIC relation builds on Parnas' *USES* relation by changing the call action to only call the next layer if and only if the TVF is true.

**Relation 2 (Chaining Layered Integrity Checks)**

$CLIC(L_j, L_i) = iff\ TVF(L_j)$ *and* $L_i$ *calls* $L_j$ *where* $j = i + 1$

Having defined the CLIC relation, I will now prove that in a composed system the system's integrity is valid when the CLIC relation holds.

**To Prove:** *That the integrity of a system with $n$ layers, $S_n$ where $CLIC(L_i, L_j)$ holds, is valid-where $L_i$ and $L_j$ are both layers in $S_n$ and $j = i + 1, \forall i, j : i < n$.*

**Proof (By Induction over n):**

**Base Case:** $n = 1$

*In this case, the system, $S_1$, has only one layer- $L_0$. By the base and physical security properties, the integrity of $L_0$ is valid and thus $S_1$ is valid as well.*

**Induction Step:**

*Assume that the integrity of a system with $k$ layers, $S_k$, is valid, and $CLIC(L_i, L_j)$ holds $\forall i, j : j = i + 1, i < k$.*

*To show that the integrity of a system, $S_{k+1}$, is valid with $k + 1$ layers, where $CLIC(L_i, L_j)$ holds $\forall i, j : j = i + 1, i < (k + 1)$.*

*Now compose a system, $S_{k+1}$, of $k+1$ layers by adding a new layer, $L_{k+1}$, to $S_k$. The new layer is either within the execution path of the system in which case $USES(L_k, L_{k+1})$ holds or the layer is orphaned and $USES(L_k, L_{k+1})$ does not hold. The former case is all that we are concerned about since if the layer is orphaned it can be (and should be) removed from the system without loss of functionality. Since $USES(L_k, L_{k+1})$ holds, $CLIC(L_k, L_{k+1})$ also holds by assumption. Thus $TVF(L_{k+1})$ is true, and therefore by the definition of the transition validation function, the integrity of $L_{k+1}$ is valid. Further by the dynamic integrity property, the integrity of the levels $L_1$ through $L_k$ are guaranteed to remain valid.*

*By the definition of $S_k$ and the definition of composition, $S_{k+1} = S_k + L_{k+1}$. The integrity of $L_{k+1}$ was previously shown, and the integrity of $S_k$ is assummed. Therefore, the integrity of $S_{k+1}$ is valid as well since the catenation of two objects whose integrity is valid results in an object whose integrity is valid.*

*Thus by the induction hypothesis principle, the integrity of a system, $S_n$, where $CLIC(L_i, L_j)$ holds is valid, where $L_i$ and $L_j$ are both layers in $S_n$ and $\forall i, j : j = i + 1, i < n$* $\Box$

# Chapter 5

# AEGIS Secure Bootstrap Architecture

This chapter describes an embodiment of CLIC- AEGIS a secure bootstrap process. AEGIS is the first complete implementation of a secure bootstrap using commercial off the shelf (COTS) hardware. I first state AEGIS' goals, and the assumptions that must be made about the system. The chapter ends by describing the ideal architecture, AEGIS, for a secure bootstrap for the IBM personal computer.

## 5.1 Goals

The goals of the AEGIS architecture are simple, but important. The primary goal is to provide a high integrity bootstrap process for a computer system by applying CLIC. This protects the system from executing malicious or damaged components ensuring that the system continues to operate as expected, *i.e.* the system remains "trustworthy". A secondary goal of AEGIS is to use the mechanism provided by CLIC to ease the burden of administering a large number of computer systems by permitting secure automatic updates of bootstrap components, error reporting, and recovery.

## 5.2 Assumptions

The architecture for every security system requires the architect to make certain design assumptions. The key is to limit the number and size of these assumptions to a minimum. The assumptions that AEGIS requires are:

**Assumption 5.1** *The hardware comprising the system is "trustworthy".*

**Assumption 5.2** *The physical security of the hardware is protected through a comprehensive physical security program.*

**Assumption 5.3** *The integrity of a small portion of the firmware, e.g. BIOS, remains valid.*

**Assumption 5.4** *A public key infrastructure (PKI) and an associated certificate authority exists.*

The underlying hardware comprises the machine. Higher layers in the hierarchy create "virtual machines" on top of the physical machine in order to ease programming and add functionality. CLIC shows that the validity of higher layers in the hierarchy are dependent upon the validity of lower layers. Therefore, the validity of the hardware layer must either be demonstrated, as in CLIC, or assumed. In AEGIS, we assume the validity of the hardware layer. This assumption is reasonable since the hardware usually does not contain any mutable storage visible to the higher layers[1].

Because firmware provides the AEGIS functionality, the integrity of a small portion of the firmware must be assumed. This prevents a "Chicken and the Egg" problem where the code validates itself. Ideally, the functionality provided by AEGIS should be moved into the hardware

[1]If future processors operate as in the Pentium Pro and Pentium II microcode updates, then these updates can be validated before installation.

layer via a cryptographic coprocessor, or cryptographic primitives on the main processor, and key storage.

Finally, the existence of a PKI must be assumed in order to support the use of digital signatures. Fortunately, a large number of different PKI's exist today that can be leveraged to provide the required infrastructure.

## 5.3 Ideal Prototype Architecture

This section describes an "Ideal" architecture for the IBM personal computer assuming that the BIOS could be completely re-engineered and small modifications made to the hardware of the motherboard.

AEGIS modifies the boot process shown in Figure 2.1 so that all executable code, except for a very small section of trusted code, is verified prior to execution by using a digital signature. This is accomplished through modifications to the BIOS, and the addition of a network card containing a flash ROM. The BIOS contains the cryptographic code and the root public key certificate.The flash ROM on the network card contains the recovery code and possibly backup copies of the bootstrap components. The network board also contains code that allows the secure recovery of any integrity failures found during the initial bootstrap. A small section of the BIOS and the network card flash ROM will comprise the only "trusted" software in the AEGIS prototype. In essence, this "trusted" software serves as the root of an authentication chain that extends to the operating system and potentially beyond to application software [PG89] [GDM89] [Mic]. A high level depiction of the AEGIS bootstrap process is shown in Figure 5.1. In the AEGIS boot process, either the operating system kernel is started, or a recovery process is entered to repair any integrity failure detected. Once the repair is completed, the system is restarted if required by the repair, *e.g.* re-flash of BIOS, or control is passed to the repaired/shadowed component. This entire process occurs without user intervention.

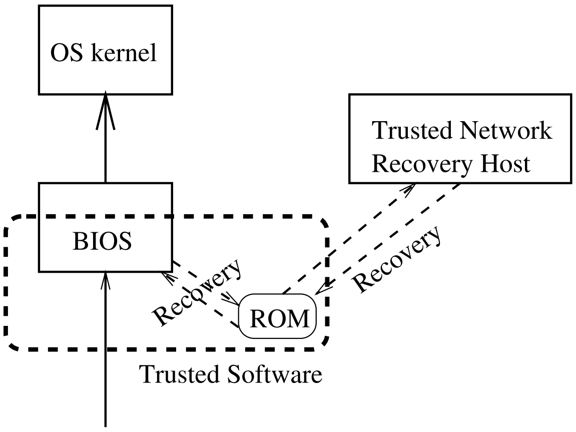


Figure 5.1: AEGIS boot overview

In addition to ensuring that the system boots in a secure manner, AEGIS can also be used to maintain the hardware and software configuration of a machine. Since AEGIS maintains a copy of the certificate for each expansion card,[2] any additional expansion cards will fail the integrity test. Similarly, a new operating system can not be started since the boot block and kernel would both change and no longer match the stored device certificate.

### 5.3.1 AEGIS Layering

We have divided the boot process (see section 2.2) into several levels to simplify and organize the AEGIS BIOS modifications, as shown in Figure 5.2. Each increasing level adds functionality to the system, providing correspondingly higher levels of abstraction. The lowest level is Level 0. Level 0 contains the *trusted* software, digital signature certificates, public key certificates, and

[2]Ideally, certificates would be embedded in the ROM by the manufacturer.

recovery code on the network card[3]. The integrity of this level is assumed to be valid. AEGIS does, however, perform an initial 16-bit additive checksum test to identify ROM failures. Additionally, this layer must be stored on write protected storage, *e.g.* the boot block of a flash ROM.

The first level contains the remainder of the usual BIOS code. The second level contains all of the expansion ROM cards and their associated ROMs, if any. The third level contains the operating system boot block(s). These are resident on the bootable device and are responsible for loading the operating system kernel. The fourth level contains the operating system, and the fifth and final level contains user level programs and any network hosts.

The transition between levels in a traditional boot process is accomplished with a jump or a call instruction without any attempt at verifying the integrity of the next level. AEGIS, on the other hand, uses digital signature certificates to protect the transition from each level to the next higher one, and its recovery process ensures the integrity of the next level in the event of failures.

The resultant AEGIS boot process is shown in Figure 5.2. Note that when the boot process enters the recovery procedure it becomes isomorphic to a secure network boot, with the difference that AEGIS only attempts to recover the component that needs repair, not the operating system as in a secure network boot.

### 5.3.2 AEGIS Actions on Failure

#### 5.3.2.1 Network Recovery

The remainder of this section describes the protocol that an AEGIS client and a trusted repository will use to mutually authenticate and transfer components and certificates.

[3]The code contained on the network card is verified prior to execution. It is contained in Level 0 because of its importance to the recovery process.

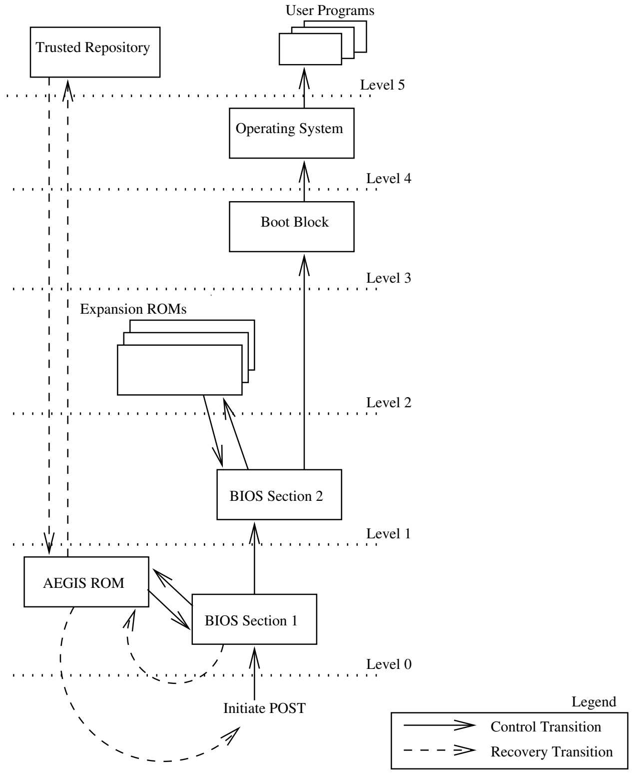


Figure 5.2: AEGIS boot control flow

A Client (AEGIS) and a Server (Trusted Repository) wish to communicate and establish a shared secret after authenticating the identity of each other. There has been no prior contact between the Client and the Server other than to agree on a trusted third party, or a public key infrastructure, to sign their authorization certificates, $C_{AR}$. The Server and the Client also need to have a copy of the trusted third party's public key, $P_{CA}$. The Client sends a message to the Server containing the Client's authorization and authentication certificates, $C_{AR}, C_{AN}$. The Server receives the message and verifies the Client's signature on the authentication certificate and that the hash contained in the authentication certificate matches that of the message, $M$. The signature of the Certificate Authority (CA) on the authorization certificate is also verified. If all are valid and the timestamp on the authentication certificate is within bounds, then the Server sends to the Client a message containing its authorization and authentication certificates. The server's authentication certificate may include the optional DH parameters, $g$ and $p$, where $Y = g^y \mod p$. If the DH parameters are not included in the certificate, then default values for $g$ and $p$ are used. Currently, AEGIS uses the same default values as those used in SKIP[AMP]. The server's nonce, $snonce$, and the client's nonce, $cnonce$, are also included in the message. The Client receives this message and verifies the signatures on the authentication and authorization certificates, that the hash in the servers authentication certificate matches the message hash, and that $cnonce$ matches the one sent in the first message. If all are valid and the timestamp value of the authentication certificate is within bounds and $cnonce$ matches that sent in the first message, then the Client sends a signed message to the Server containing its DH parameter $X$, where $X = g^x \mod p$, and the server's nonce $snonce$. The Server receives the message and verifies both the signature and that $snonce$ matches that sent in its previous message. If both are valid, then the Server can generate the shared secret, $k$, using DH, $k = X^y \mod p$. The Client similarly generates the shared secret, $k = Y^x \mod p$. The shared secret, $k$, can now be used to authenticate messages between the Server and the Client until such time as both agree to change $k$. Figure 5.3 depicts the entire exchange between the Client and the Server. In Figure 5.3, $V_p(S)$ represents the verification of the signature, $S$, with the public key belonging to the principal $p$. $S_p(M)$ represents the signing of message $M$ using the private key belonging to principal $p$.

The authentication certificate assists in ensuring that the protocol is "Fail Stop" through the use of nonces and a short validity period for the certificate. The use of $snonce$ also permits the Server to reuse $Y$ over a limited period[GS95], reducing the computational overhead on the server during high activity periods. The potential for a TCPSYN like denial of service attack[HB96] is mitigated in the same manner by the authentication certificate. The authorization certificate also prevents clients from masquerading as a server because of the client/server capability tag. This is

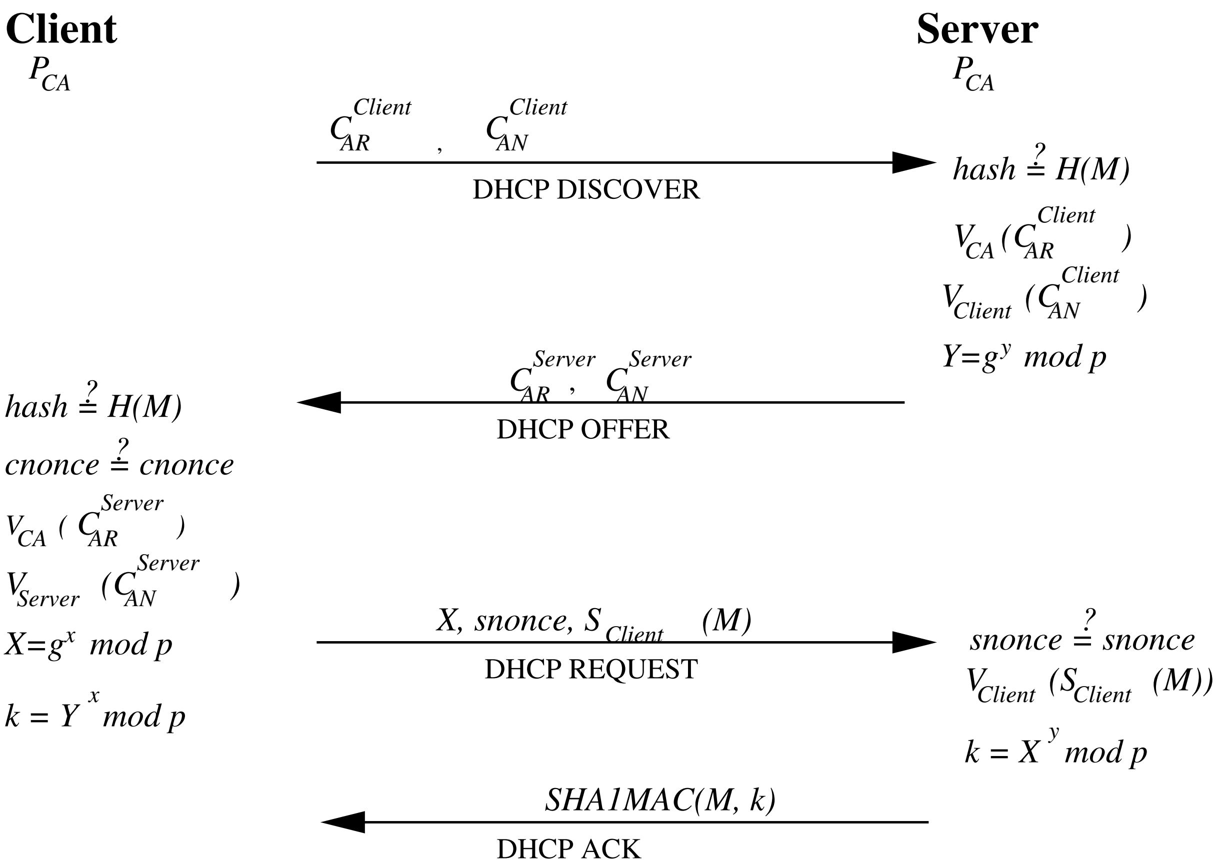


Figure 5.3: Authentication Message Exchange

| *Unique Device ID (14 bytes)* | *Abs. Loc. (4 bytes)* | *Size (4 bytes)* | *Reserved (10 bytes)* |
|---|---|---|---|

Figure 5.4: Recovery ROM Component Table Entry Format

a benefit not possible with X.509 based certificates.

Subsequent messages, *e.g.* TFTP messages, use a SHA1 message authentication code (MAC) augmented with a one up counter to prevent replays[KBC97]. The counter is initially set to zero when the shared secret, $k$, is derived.

#### 5.3.2.2 ROM recovery

The recovery ROM can reside either on a network interface card (NIC) or a ROM card. Currently, the recovery ROM resides on an Intel eePro100B PCI network interface. Since the purpose of NIC ROM is to store recovery components rather than executable content, the format does not have to comply with the normal PCI expansion ROM format [SW96]. I do use the standard format, however, to leverage off of the PCI ROM validation code. This requires the first, and only, instruction contained in the ROM to be a return far instruction.

Following the return far instruction is a table listing the components available in the ROM by their unique id (see Section 6.1.1). The exact format of an entry in the table is shown in Figure 5.4. The last table entry contains all zeros to indicate the end of the table.

### 5.3.3 Infrastructure and Administration

The proper administration of a system is paramount to its proper operation and security. Estimates are that over 90% of all system penetrations on the Internet are because of improper administration

| Component Type | Storage Location |
|---|---|
| BIOS | Embedded in vendor specific location |
| Expansion ROM | Embedded after ISA/PCI header |
| Primary Boot Loader | Embedded at end |
| Secondary Boot Loader | Embedded at end |
| Kernel | Embedded in header |

Table 5.1: Ideal Certificate Storage Locations

of the system. Therefore, careful attention must be paid to the administration of AEGIS because of its role in the bootstrap process. While AEGIS is still a prototype, several administrative aspects must be addressed. The first is the creation of the certificates for the bootstrap components. Ideally, this would be done by the manufacturer or vendor of the component and embedded within the component itself as in Authenticode. Until such time as vendors support the signing of components as in Authenticode [Mic], the tools and a procedure must be provided to system administrators to create and install the signatures. The remainder of this section will describe an ideal environment where the vendors sign their components. Section 6.1.6 presents the administration details for the prototype.

Ideally, a component's certificate is embedded within the component itself, and is signed by an authority whose ancestry can be traced directly to the public key embedded in the AEGIS firmware. Table 5.1 shows the certificate storage location for each type of component. Embedding the certificate in the component permits the most flexibility in AEGIS since the system does not need to store the certificates itself, and therefore the number of supported components is potentially limited. It also significantly eases the administrative burden since local users no longer need to sign components- unless the local authorities also wish to support specific configurations.

### 5.3.4 Security Analysis

The ideal architecture has two vulnerable points, each of which occurs by invalidating an assumption. By violating the physical security of the computer system, the base layer or the underlying hardware could be modified so that a malicious user could add and run unauthorized components during the boot process, corrupting the entire process. The second vulnerable point involves the inadvertent approval of a malicious layer by either malice or careless verification. This malicious layer could then violate the dynamic integrity, *i.e.* modify itself or the lower layers, of the system. Fortunately, both problems can be mitigated to a large degree through enforcement of robust policies for physical security and code vetting.

# Chapter 6

# Implementation and Experimental Results

This chapter describes the implementation of the AEGIS prototype and how the details differ from the ideal architecture presented in Chapter 5. I conclude the chapter by presenting the cost, from a user's perspective, of using AEGIS in a desktop environment, along with detailed timing information of the verification cost of the bootstrap components.

## 6.1 Implementation

The implementation of the AEGIS prototype was accomplished by modifying BIOS source code purchased from Microid Research. The source code supports most of the Pentium motherboards using the Intel 430 HX PCI chipset. While the source code supports most HX chipset motherboards, the prototype was built and tested on an ASUS P55T2P4 Pentium motherboard. The prototype has run successfully on this motherboard with both Pentium and AMD K6 processors.

Numerous tools were used to build the prototype. The original BIOS source code required Microsoft assembler (MASM) version 5.11d, and a version of the linker other than the one bundled with MASM 5.11d. The source was ported to the most recent version of MASM by Harold Bower. The linker, however, could not be changed because a "feature" of the newer Microsoft linkers could not fix-up segments with mixed 16-bit and 32-bit code. The AEGIS additions to the BIOS were written in assembler using NASM, and MASM, and also in C, using Microsoft Visual C++ version 5.0. The object files created by the assemblers and compiler were linked together to produce a "stand-alone" program using Beacon's Link and Locate LL386 linker. LL386 was used for two reasons. The first was that it supports the linking of 16-bit and 32-bit segments. The second reason, and the most important, was that LL386 can produce an OMF386 boot loadable binary [Cor85]. An OMF386 boot loadable binary *was* required to support source level debugging using the American Arium ECM-55 hardware In-circuit emulator.

The remainder of this section presents the implementation details of the AEGIS prototype. The second section presents the detailed timing information on the prototype.

### 6.1.1 Digital Signature Certificate

The format of the digital signature certificate used in AEGIS is a variation of the SDSI certificate format (see Section 2.6). Both X.509 and SDSI certificates could not be used directly because of the storage size of the code implementing them. The certificate code for AEGIS currently requires approximately 2KB (uncompressed) of ROM space, significantly smaller than the over 30KB required for X.509 support.

The layout of the AEGIS digital signature certificate was designed to save space without forfeiting required elements such as identity and expiration time. The first entry in the certificate is the

| Type | Resvd | Vendor | Device | Rev | Proj | Sub | Base | Month | Day | Year |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |

| |
|---|
| Device Hash<br>16 bytes |
| Public Key ID<br>16 bytes |
| Signature<br>128 bytes |

Figure 6.1: AEGIS Digital Signature Certificate

| Component Type | ID value |
|---|---|
| F000 Segment | 0 |
| E000 Segment | 1 |
| ISA ROM | 2 |
| PCI ROM | 3 |
| Primary Boot loader | 4 |
| Secondary Boot loader | 5 |
| Kernel | 6 |
| Public Key | 7 |

Table 6.1: Component Types

unique identity (principle in SDSI). The identity is the same as that used with PCI devices [SW96] (see Figure 6.1). The first byte of the certificate represents the type of component over which the signature was made, Table 6.1 shows the various types and their values. The next byte in the certificate is currently reserved for alignment purposes[1]. The next eight bytes are identical to the header information in a PCI ROM. When signing a PCI ROM, this information can be used directly without changes. When signing another component, the values must be derived in a deterministic manner. The next four bytes indicate the expiration time of the certificate. The certificate expires at midnight Greenwich mean time (GMT) on the date indicated. The values for the month, date,

[1]MSVC could not agree with Watcom and gcc on alignment.

| Boot Component | Arguments |
|---|---|
| BIOS Segments | None |
| ISA ROM | ROM address |
| PCI ROM | ROM address |
| Boot Loader | Loader address |
| | Number of 512 byte sectors |
| | Certificate address |

Table 6.2: AEGIS arguments

and year are stored in their hexadecimal representation. As a result, AEGIS is Y2K compliant. Following the expiration fields is the MD5 hash of the signed component, and the MD5 hash of the signer's public key. The last field of the certificate is the actual signature over the first 46 bytes of the certificate. The resultant certificate is 174 bytes in length for a certificate of depth one, *i.e.* the certificate is signed by the certificate authority.

### 6.1.2 BIOS Modifications

The number of changes to the base BIOS source code were kept as small as possible. The primary reason for this were to make the AEGIS modules as portable as possible across other BIOS implementations. As a result, AEGIS only requires the placement of "hooks" at the appropriate locations (prior to passing control to a new component) during the POST, and the AEGIS software placed within the BIOS- ideally in the boot block of the flash ROM.

An AEGIS hook has three steps. The first step is to ensure the AEGIS code is resident in RAM, and that AEGIS interrupt service routine (ISR) has been established. The second step is to place arguments identifying the boot component to be checked, *e.g.* ISA ROM, along with location (address) and size information into the appropriate registers, see Table 6.2. The final step is to issue a 0xC3 software interrupt that causes execution to jump into the AEGIS initialization code,

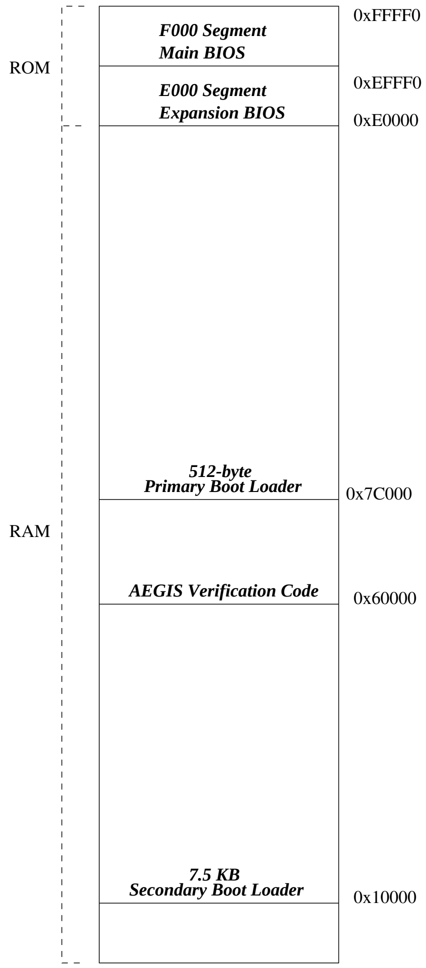


Figure 6.2: Memory map for AEGIS prototype

see Section 6.1.3.

The storage location for the current AEGIS code is, unfortunately, not in the boot block. Restructuring the BIOS source code so that AEGIS would fit in the boot sector area (0xF0000 - 0xF2000) proved too difficult given the time constraints of this dissertation. Currently, the location of the AEGIS code is within the 112KB main block of the flash ROM. This area is also mapped into main memory as the 0xE000 segment. A memory map for AEGIS is shown in Figure 6.2.

### 6.1.3 AEGIS code

The entry point for AEGIS is currently fixed at address 0x60000. AEGIS initializes itself by saving the current processor state and transitioning from real mode (16bit) to protected mode (32bit), see Appendix 9. The added cost of using protected mode, rather than "big real mode", is valuable due to the memory protection provided by protected mode. Once in protected mode the text, data, and stack segments are established, and the BSS segment zeroed.

The type of the component to be checked is then determined by the argument passed in the BL register, and the appropriate verification routine called, see Appendix 11. The verification routine checks the validity of the signature certificate and compares the hash contained in it to a computed hash. If the two match, then a zero value is returned. If they do not match, then a non-zero value is returned. If both the certificate and the component are valid, then AEGIS transitions back into real mode and restores the state of the processor. If either the digital certificate or the component are invalid, then a recovery procedure is entered.

### 6.1.4 PicoBSD Boot loader modifications

PicoBSD, as a variant of FreeBSD 3.0, uses a primary and a secondary boot loader. The current AEGIS prototype only supports a boot floppy because of efforts to develop a secure active network node, see Section 7.1. The primary loader is 512 bytes in length and resides on sector 0 of a boot floppy. While the PicoBSD primary loader has room to store its own signature certificate, most other primary loaders, *e.g.* MS-DOS, do not. As a result, the signature certificates for authorized primary loaders are stored in the E000 segment of the BIOS. Secondary loaders are at least 7 KB, and as a result their signature certificates can be embedded.

The changes in the primary boot sector were to pass control to the AEGIS code with the appropriate arguments set. The changes to the secondary loader were more significant since transitions between protected and real mode were needed, as well as code to determine the size of the kernel's text and data segments. The modified PicoBSD loaders, with the changes highlighted, are shown in Appendix 12.

### 6.1.5 Recovery

The current AEGIS prototype does not implement the network recovery protocol as described earlier (see Section 5.3.2). Hard-to-remedy incompatibilities between the tools used to build the BIOS and the initial network recovery prototype prevented integrating the two prototypes in a timely fashion. The current AEGIS prototype does, however, implement the recovery of invalid components via either an ISA or PCI expansion ROM (see Section 5.3.2.2). While this approach is less functional than network-based recovery, it is sufficient as a proof of concept for the recovery aspects of CLIC.

The current prototype implements the ROM recovery mechanism described earlier in Section 5.3.2.2.

### 6.1.6 Administration

Good administration of a system is essential to the system's security. The current AEGIS prototype includes several utilities for administrators to sign boot components. These utilities are available on a single MS-DOS boot floppy that creates the signature certificates and inserts them in the proper locations. The source code for one of these programs, isascan, is shown in Appendix 13. Isascan scans the ISA bus and identifies and computes a digital signature for any expansion ROM found.

### 6.1.7 Security Analysis

Due to several constraints, the current AEGIS prototype does not completely match the ideal architecture described previously. This section, briefly, analyzes the security impact of those differences.

The only significant security problem with the current prototype is that the base layer does not reside on the boot block of the BIOS flash ROM. As a result, the contents of the base layer, *i.e.* AEGIS code, can be modified without violating the physical security of the system. Placing the base layer in the boot block of the flash ROM would have required a major rewrite of the core BIOS source code. This was, unfortunately, beyond the scope of the dissertation. An attempt was made to replace the current i28F001BX-T with a i28F001BX-B (see Figure 2.6) so that the base layer could be protected. Unfortunately, this attempt failed.

## 6.2 Experimental Results

This section presents the results of the AEGIS prototype implementation. A user can not determine the difference between a system protected with AEGIS and one that isn't- provided that recovery is not required. When recovery is required, the overhead added by a ROM recovery is minimal[2]. The overhead added by a network recovery is larger, but still small compared to the loss of use system.

### 6.2.1 Bootstrap

The overhead associated with AEGIS is insignificant when amortized into the total cost of the bootstrap of a personal computer. This section compares the cost of a non-AEGIS bootstrap with an AEGIS bootstrap (without recovery) for the ASUS P55T2P4 motherboard with a 166 Mhz Pentium processor, sixty four megabytes of memory, a sixty four kilobyte PCI video ROM, and

[2]Provided a reboot is not required.

| Component Type | Size | Cycle count | Time in seconds |
|---|---|---|---|
| BIOS Segments | 2 x 64 KB | 0x2E5E702B | 4.686 |
| PCI Video ROM | 64 KB | 0x101F804B | 1.629 |
| Primary Boot Loader | 512 bytes | 0x0091B40E | 0.057 |
| Secondary Boot Loader | 7 KB | 0x00969CEC | 0.059 |
| Kernel Text and Data | 1.2 MB | 0x1214e24C | 1.827 |

Table 6.3: Verification Times

a one hundred twenty eight kilobyte NIC ROM. The times[3] and cycle count for each bootstrap component are shown in Figure 6.3. The time for each figure includes the cost of switching from real mode to protected mode and back again. The cycle counts were obtained by using the RDTSC timestamp instruction available on Intel Pentium and above computers. The RDTSC instruction returns the number of clock cycles since reset in the EDX:EAX registers. The code computing the cycle counts is included in the initialization code shown in Appendix 9. The timing code is included in the assembly when the TIME macro is defined.

The total wall-clock time required to start a picoBSD kernel using AEGIS is 84 seconds. This compares favorably to starting the picoBSD kernel without AEGIS which requires 70 seconds. The pie chart shown in Figure 6.3 shows the break down of the costs. The AEGIS setup cost of seven seconds includes the cost of copying the AEGIS code from ROM into RAM and uncompressing the image. This is done three times. Thus, the overall cost of the copy and uncompress is approximately 2.3 seconds per use which is as expected given the cost of reads from ROM. The verification time for the BIOS segments and the PCI ROM are high for the same reason, the cost of reading from ROM rather than RAM. The times for the remaining components, boot loaders and kernel, are as expected.

[3]Times are for a 166 MHZ Pentium Processor.

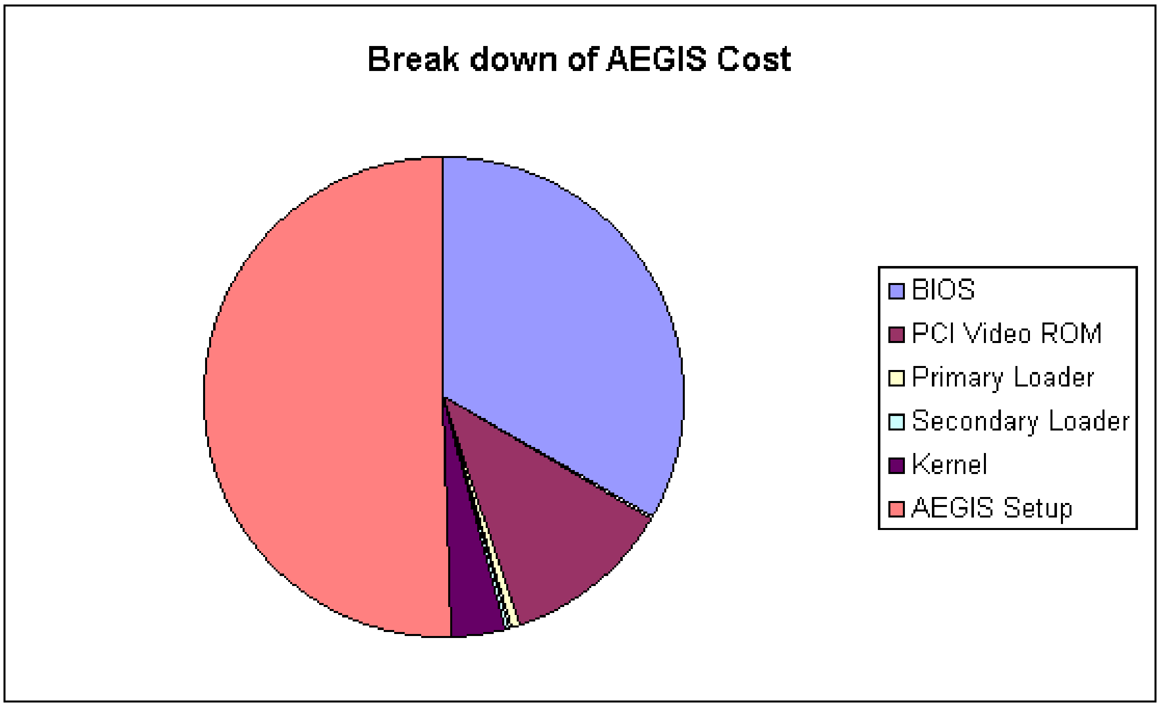


Figure 6.3: AEGIS Verification Cost

### 6.2.2 Recovery

This section describes the experimental results obtained from timing the recovery process for both the ROM and network prototypes.

#### 6.2.2.1 ROM

The time needed to perform a ROM recovery is insignificant. The time includes the cost of an additional verification to ensure the copied component is valid and the cost of a memory copy. In the case of the repair of the kernel or a boot block, then the time also includes the cost of transition to real mode to use the lower level BIOS interrupt, int 0x13, to write the new component to the disk.

The cost of recovering and repairing the 512 byte primary picoBSD boot loader is approximately[4] 3.2 seconds. This major cost of this repair is the cost of writing the new sector to the floppy drive, and a transition to real mode to use the BIOS interrupt 0x13 and the cost of the transition back to protected mode.

#### 6.2.2.2 Network

The network recovery prototype was not fully integrated with the AEGIS BIOS due to tool incompatibilities. However, some performance information could be gathered by using the recovery prototype in a *stand-alone fashion*.

The ideal protocol[5] described in Section 5.3.2 coupled with use of the X.509v3 certificate format produces packets that exceed the MTU of a local area network. As a result, the protocol was modified so that the packets would not exceed the LAN MTU. The only change was to eliminate

[4] Detailed timings are currently being taken.

[5] The network recover prototype uses DSA rather than the RSA algorithm used with the BIOS.

| Message | Size in bytes |
|---|---|
| Discover | 901 |
| Offer | 1081 |
| Request | 626 |
| ACK | 626 |

Table 6.4: DHCP packet sizes

| Message | Time in ms |
|---|---|
| Discover | 153.3 |
| Request | 40 |

Table 6.5: DHCP client message performance

the use of the authentication certificate and replace it with a signature over the entire packet. This results in the following packet sizes using an authorization certificate of depth one (see Table 6.4).

The performance for the two client messages sent from a Pentium 266 Mhz is shown in Table 6.5, and Figure 6.4. The cost of the DISCOVER message is significant. The primary reason for this is excessive paranoia on my part with respect to the random number generator. As Figure 6.4 shows, the total cost of seeding the random number generator is 149.9 ms. This number could be improved, however, by using a less paranoid seeding process. The remainder of the cost of the DISCOVER message is spent signing the packet, 34 ms.

The costs of the two server messages are similar to those of the client except that the server can rely on a stronger source of entropy. The costs of two server messages are shown in Table 6.6, and the cost of an OFFER message is broken down in Figure 6.5.

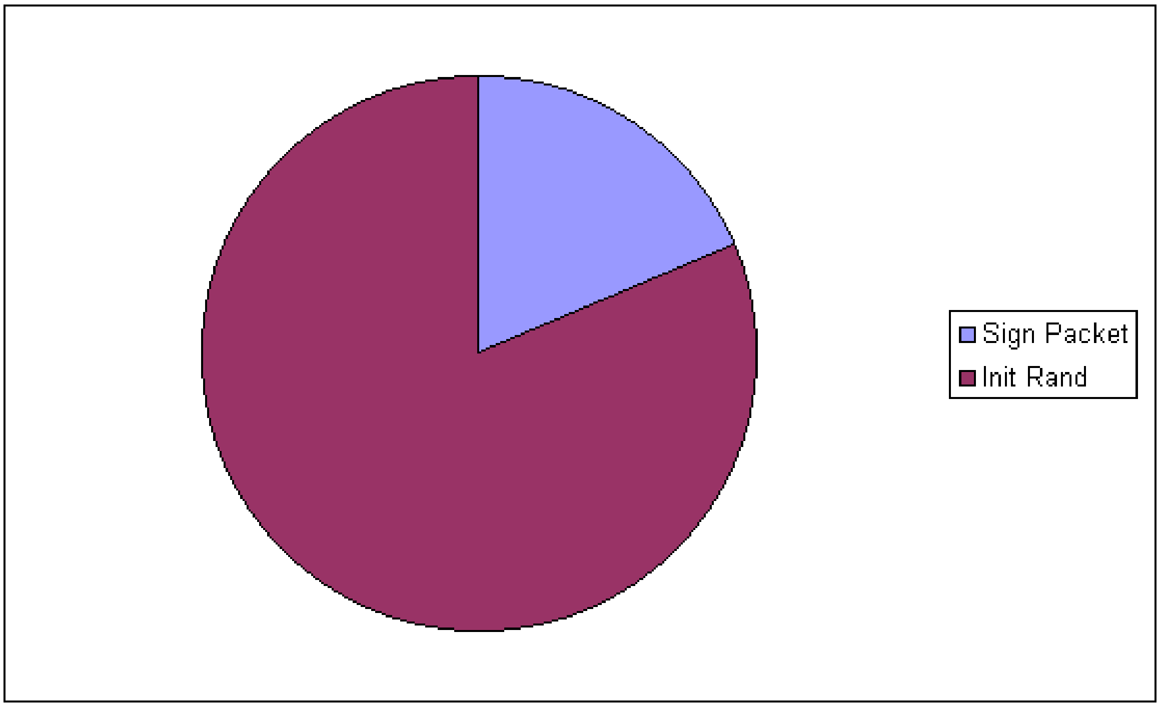


Figure 6.4: DHCP Discover Message Cost break down

| Message | Time in ms |
|---|---|
| Offer | 221 |
| ACK | 126 |

Table 6.6: DHCP server message performance

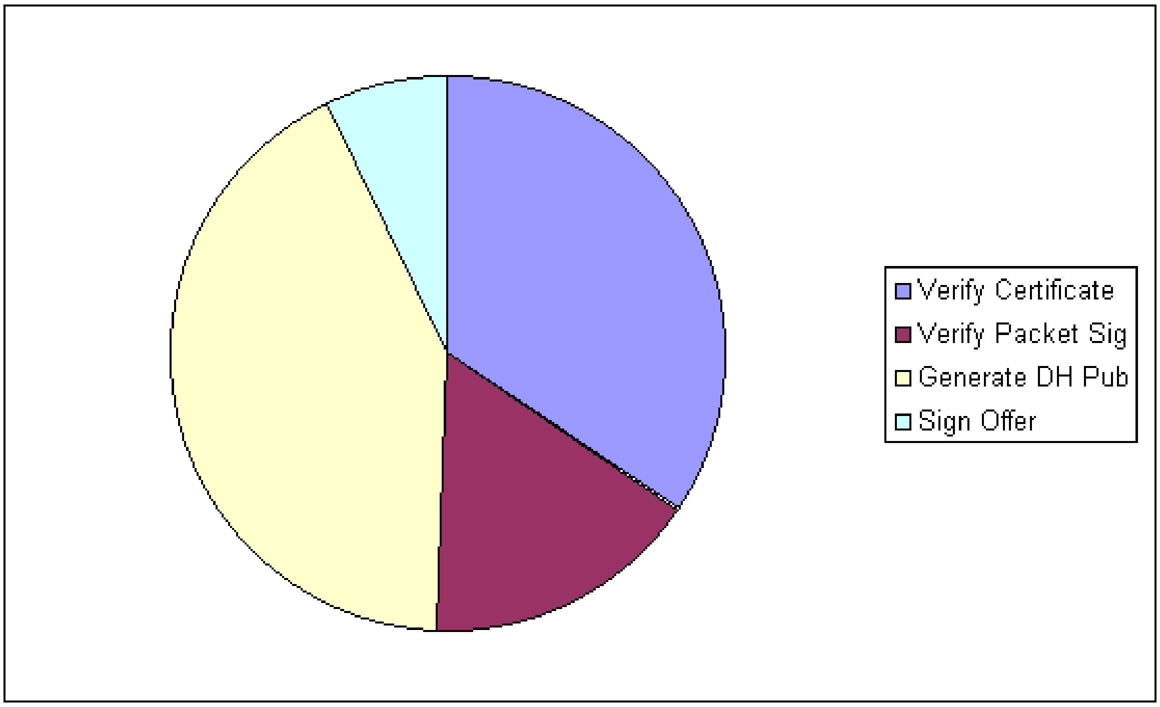


Figure 6.5: DHCP Offer Message Cost break down

# Chapter 7

# Applications and Future Work

This chapter describes how AEGIS is used to support Active Network research, and a novel architecture, SIBL, for sharing a workstation across different security policies. The chapter concludes by describing work with the Internet Engineering Task Force (IETF) to standardize an authentication mechanism for DHCP[1].

## 7.1 Secure Active Network Environment

The desire for flexible networking services has given rise to the concept of "active networks." Active networks provide a general framework for designing and implementing network-embedded services, typically by means of a programmable network infrastructure. A programmable network infrastructure creates significant new challenges for securing the network infrastructure.

This section begins with an overview of active networking. It then moves to security issues, beginning with a threat model for active networking, moving through an enumeration of the challenges for system designers, and ending with a survey of approaches for meeting those challenges.

[1]This effort is a spin-off of the recovery protocol from AEGIS.

The Secure Active Networking Environment (SANE) realizes many of these approaches; an implementation exists and provides acceptable performance for even the most aggressive active networking proposals such as active packets (sometimes called "capsules").

### 7.1.1 What is Active Networking ?

In networking architectures a design choice can be made between:

1. Restricting the actions of the network infrastructure to transport, and
2. easing those restrictions to permit on-the-fly customization of the network infrastructure.

The data-transport model, which has been successfully applied in the IP Internet and other networks, is called passive networking since the infrastructure (*e.g.,* IP routers) is mostly indifferent to the packets passing through, and their actions (forwarding and routing) cannot be directly influenced by users. This is not to say that the switches do not perform complex computations as a result of receiving or forwarding a packet. Rather, the nature of these computations cannot dynamically change beyond the fairly basic configuration options provided by the manufacturer of the switch.

In contrast, *active* networking allows network-embedded functionality other than transport. For current systems, this functionality ranges from WWW proxy caches, multicasting[Dee89] and RSVP[BZB+97] to firewalls. Since each of these independently designed and supported functions could be carried out as an application of a more general infrastructure, the architecture of such infrastructures is now being investigated aggressively.

The basic principle employed is the use of *programmability*, as this allows many applications to be created, including those un-imagined by the designers of the switch. There are a number of forms this programmability can take, including treating each packet as a program (active packets or "capsules") and programming or reprogramming network elements on-the-fly with select packets.

Note that the latter approach subsumes the former, as a program may be loaded which treats all subsequent packets as programs.

#### 7.1.1.1 Why is Active Network Security Interesting?

From a security perspective, a large scale infrastructure with user access to programming capabilities, even if restricted, creates a wide variety of novel and difficult, challenges. Most directly, since the basis of security is controlled access to resources, the increased complexity of the managed resources makes securing them much more difficult. Since "security" is best thought of as a mapping between a policy and a set of predicates maintained through actions, the policy must be more complex, resulting in an explosion in the set of predicates.

For example, the ability to load a new queuing discipline may be attractive from a resource control perspective, but if the queuing discipline can replace that of an existing user, the replacement policy must be specified, and its implementation carefully controlled through one or more policy enforcement mechanisms.

Additionally, such a scenario forces the definition of *principals* with which policies are associated. When compared with the policy at a basic IP router (no principals, datagram delivery guarantees, FIFO queuing, *etc.*) it can be seen why securing active networks is difficult.

### 7.1.2 Virtual and Real Resources

As the role of active networking elements is to store, compute and forward, the managed resources are those required to store packets, operate on them, and forward them to other elements. The resources provided to various principals at any instant cannot exceed the real resources (*e.g.,* output port bandwidth) available at that instant. This emphasis on real resources and time implies that a conventional <*object, principal, access*> 3-tuple for an access control list (ACL) is inadequate.

To provide controlled access to real resources, with real time constraints, a fourth element to represent duration (either absolute or periodic) must be added, giving <*object, principal, access, QoS guarantees*>. This remains an ACL, but is not "virtualized" by leaving time unspecified and making "eventual" access acceptable. We should point out that this new element in the ACL can be encoded as part of the *access* field. Similarly, we need not use an actual ACL, but we may use mechanisms better-suited for distributed systems which can be expressed in terms of ACLs.

### 7.1.3 Threat Model

An active network infrastructure is very different from the current Internet [AAKS98]. In the latter, the only resources consumed by a packet at a router are:

1. the memory needed to temporarily store it, and
2. the CPU cycles necessary to find the correct route.

Even if IP [Pos81] option processing is needed, the CPU overhead is still quite small compared to the cost of executing an active packet. In such an environment, strict resource control in the intermediate routers was considered non-critical. Thus, security policies [Atk95] are enforced end-to-end. While this approach has worked well in the past, there are several problems. First, denial of service attacks are relatively easy to mount, due to this simple resource model. Attacks to the infrastructure itself are possible, and result in major network connectivity loss. Finally, it is very difficult to provide enforceable quality of service guarantees. [BZB$^+$97]

Active Networks, being more flexible, considerably expand the threat possibilities. The security threats faced by such elements are considerable. For example, when a packet containing code to execute arrives, the system typically must:

- Identify the sending network element.
- Identify the sending user.
- Grant access to appropriate resources based on these identifications.
- Allow execution based on the authorizations and security policy.

In networking terminology, the first three steps comprise a form of admission control, while the final step is a form of policing. A second view is that of static versus dynamic checking. Security violations occur when a policy is violated, *e.g.,* reading a private packet, or exceeding some specified resource usage.

#### 7.1.3.1 Denial of Service Attacks

Cryptographic mechanisms have proven remarkably successful for functions such as identification and authentication. These functions typically (although not necessarily) are used in protocols with a *virtual time* model, which is concerned with sequencing of events rather than more constrained sequencing of events with time limits (the *real time* model). The cases where time limits are observed are almost always for reasons of robustness, *e.g.,* to force eventual termination. Since such timeouts are intended for extreme circumstances, they are long enough so that they can cope with any reasonable delay.

In an environment where a considerable fraction (and perhaps eventually a majority) of the traffic will be continuous media traffic, security must include resource management and protection with an eye to preserving timing properties. In particular, a pernicious form of "attack" is the so-called "denial of service" attack. The basic principle applied in such an attack is that while wresting control of the service is desirable, the goal can be achieved if the opponent cannot use

```
#!/bin/sh
$0 #invoke ourselves
```

Figure 7.1: **A recursive shell script for UNIX**

the service. This principle has been used in military communications strategies, *e.g.,* the use of radio "jamming" to frustrate an opponent's communications, and most recently in denying service to Internet Service Provider servers using a TCP *SYN* flood attack [pan96, dri96]. Another very effective (even crippling) attack on a computer system can occur due to scheduling algorithms which implicitly embed design assumptions.

To look at an example in some detail, consider the so-called "recursive shell" shown in Figure 7.1.3.1.

The shell script invokes itself. This is in fact a natural programming style, except that the process of invoking a shell script consists mainly of executing two heavyweight system calls, `fork()` and `exec()`, which, respectively, create a new copy of the current process and replace the current process with a new process created from an executable file. Since the program spends the majority of its time executing system calls, which in UNIX cause the operating system to execute on behalf of the user (at high priority) the system's resources are typically consumed by this program (including CPU time and table space used for holding process control blocks).

With an active network element, it is easy to imagine situations where user programs (or errant system programs) run amok, and make the network elements useless for basic tasks. The solution, we believe, is to constrain *real* resources associated with active network programs. For example, if we limited the principal (*e.g.,* a "user") invoking the recursive shell script to 10% of the CPU time, or 10% of the system memory, the process would either limit its effects on the CPU to a 10% degradation, or fail to operate (since it could not invoke a new process) when it hit the table space limitation. Fortunately, a number of new operating systems[MMO+94, LMB+96] have appeared which provide the services necessary to contain one or more executing threads within a single scheduling domain.

### 7.1.4 Challenges for the System Designer

Independent of the specific network architecture, the designer of a network has a set of tradeoffs they must make which define a "design space." We consider five here:

1. *Flexibility*. Flexibility is a measure of the system to perform a variety of tasks.
2. *Usability*. Usability is a measure of the ease with which the system can be used for its intended task(s).
3. *Performance*. The system will have some quantitative measures by which it is evaluated, such as throughput, delay, delay variation.
4. *Cost*. A networking system will have quantifiable economic costs, such as costs for construction, operation, maintenance and continuing improvements.
5. *Security*. Since network systems are shared resources the designer must provide mechanisms to protect users from each other according to a *policy*.

It is our belief that, as in this list, security is often left until last in the design process, which results in an under-emphasis (and consequent inattention) paid to security. If security is *designed in*, it can simply be made part of the design space in which we search for attractive cost/performance tradeoffs. For example, if acceptable flexibility requires down-loadable software, and acceptable security means that only trusted down-loadable software will be loaded, our cost and performance optimizations will reflect ideas such as minimizing dynamic checks with static pre-checks or other means. If security is not an issue, there is no point in doing this.

The designer's major challenge is finding a point (or set of points) in the design space which is acceptable to a large enough market segment to influence the community of users. Sometimes this is not possible; the commercial emphasis on forwarding performance is so overwhelming that concessions to security slowing the transport plane are simply unacceptable. Fortunately, organizations have become sufficiently dependent on information networks that *security does sell*.

In the context of active networks, the major focus of security is the set of activities which provide flexibility; that is, the facility to inject new code "on-the-fly" into network elements. To build a secure infrastructure, first, the infrastructure itself (the "checker") must be unaltered. Second, the infrastructure must provide assurance that loaded modules (the dynamic checking) will not violate the security properties. In general, this is very hard. Some means currently under investigation include domain-specific languages which are easy to check (*e.g.*, PLAN), proof-carrying code [NL96b, Nec97], restricted interfaces (ALIEN), and distributed responsibility (SANE). Currently, the most attractive point in the design space appears to be a restricted domain-specific language coupled to an extension system with heavyweight checks. In this way, the frequent (per-packet) dynamic checks are inexpensive, while focusing expensive scrutiny on the extension process. This idea is manifest in the SwitchWare active network architecture[AAH+98].

### 7.1.5 SANE Architecture

Previous attempts at system security have not taken a holistic approach. The approaches typically focused on a major component of the system. For instance, Operating System research has usually ignored the bootstrap process of the host. As a result, a *trustworthy* operating system is started by an *un-trustworthy* bootstrap! This creates serious security problems since most Operating Systems require some lower level services, *e.g.,* firmware, for *trustworthy* initialization and operation. A major design goal of SANE[AAKS98] was to reduce the number and size of components that are assumed as *trustworthy*. A second major design goal of SANE was to provide a secure and reliable mechanism for establishing a security context for active networking. An application or node could then use that context in any manner it desired.

As stated earlier, one of SANE's major design goals is to limit the required assumptions. No practical system can avoid assumptions, however, and SANE is no different. Two assumptions are required by SANE. The first assumption that SANE makes is that the physical security of the host is maintained through strict enforcement of a physical security policy. The second assumption SANE makes is the existence of a Public Key Infrastructure (PKI). While a PKI is required, no assumptions are made as to the type of PKI, *e.g.,* hierarchical or "web of trust".[Com89, EFRT97, LR97, Zim95, BFIK98b]

The overall architecture of SANE for a three-node network is shown in Figure 7.2.

The initialization of each node begins with the bootstrap. Following the successful completion of the bootstrap, the operating system is started which loads a general purpose evaluator, *e.g.,* a Caml[2][Ler95] or Java[GJS96] runtime. The evaluator then starts an "Active Loader" which restricts the environment provided by the evaluator. Finally, the loader loads an "Active Network Evaluator" (ANE) which accepts and evaluates active packets, *e.g.,* PLAN [HKM$^+$98], Switchlet, or ANTS [WGT98]. The ANE then loads the SANE module to establish a security context with each network neighbor. Following the establishment of the security context, the node is ready for secure operation within the active network.

[2]Caml and Objective-Caml are dialects of ML[MTH90].

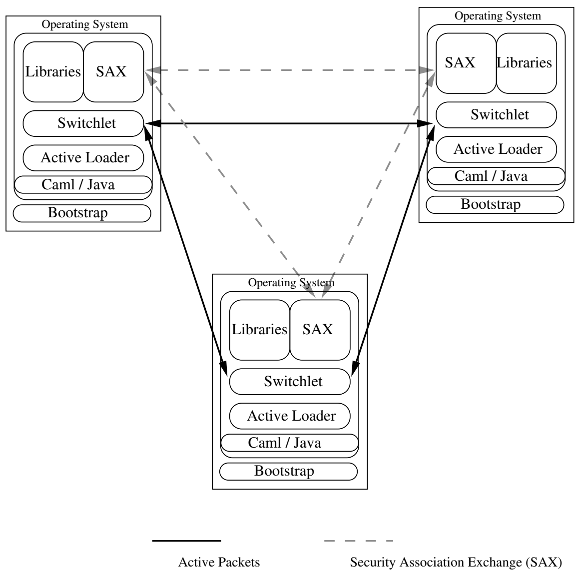


Figure 7.2: **SANE Network Architecture**

It should be noted that the services offered by SANE can be used by most active networking schemes. In our current system, SANE is used in conjunction with the ALIEN aarchitecture[Ale98]. ALIEN is built on top of the Caml runtime, and provides a network bytecode loader, a set of libraries, and other facilities necessary for active networking.

The following sections describe the three components of SANE. These include the AEGIS[AFS97, AKFS98] bootstrap system described earlier in Chapter 6, the ALIEN[Ale98] architecture, and SANE[AAH+98, AAKS98] itself.

### 7.1.6 SANE Services

SANE builds on AEGIS and ALIEN in order to provide security services for an active network. We believe that these services are minimally required for the deployment of a robust active infrastructure. This is not to say that they contain all the security mechanisms one would ever want. Rather, they are basic building blocks needed for possibly more advanced mechanisms. These services include:

- Cryptographic primitives, provided as ALIEN libraries. A number of symmetric[NBS77] and public-key[oS94] cryptosystems and hash[oS95] functions are made available for use by programmers and other system components.

- Packet authentication. This can be achieved through signing the packet with a public-key algorithm, or using some secret key method, like a MAC.

  Public-key authentication allows for zero-roundtrip authentication, since no negotiation is required, but is relatively heavy-weight computationally and is subject to some replay attacks in the absence of node-persistent state or synchronized clocks [Syv94, Gon92] in the network switches. This form of authentication is well-suited to mobile-agent types of applications, such as some active network management schemes [PJ96], or where data generated on a switch needs to be securely combined with the code.

  Secret-key based authentication is faster, and thus better suited for bulk transport. For scalability reasons however, it needs to be automated, and thus needs some public-key infrastructure to base the authentication on. The cost of the automated key establishment can be easily amortized after transmitting even a small amount of data.

- A key establishment protocol (KEP), which allows two principals in the network to establish secret keys and exchange certificates. The protocol is also used in bootstrap failure-recovery in AEGIS[AKFS98] and is based on the Station-to-Station [DvOW92] protocol, using Diffie-Hellman [DH76] key exchange and DSA [oS94] (or other) public-key signatures. This protocol is used in three different roles:

  1. Secure boostrap component recovery in AEGIS, as we discussed in Section5.3.2.

2. Secure neighbor discovery once the node boots. Similar to verifying the network cards and software components in AEGIS, at this stage the node verifies the immediate network topology and establishes trusts relations between "adjacent" switches. The shared secret keys established in this manner can be used for hop-by-hop secure transmission of data or code, which is a requirement for some mobile-agent types of applications. Other critical infrastructure information can be secured by this mechanism as well, *e.g.,* routing updates.

3. Session-key establishment, principal authentication and authorization. The shared secret keys can be used to secure the communications of any two principals in the network. Furthermore, principals can authenticate each other and establish authorizations. For example, a user can verify the identity of a switch he needs to load a program on, while the switch can determine whether the user has permission to do so, and what other restrictions apply. We make use of KeyNote[BFIK98b] credentials to provide this functionality. These credentials specify the resource usage and access control policies that ALIEN enforces.

- Administrative domains allow a set of network elements under the same administrative control to restrict security requirements when communicating with each other. In such a configuration, "border" elements act as present-day firewalls and can require *e.g.,* a number of iterations of KEP to establish all the problem credentials. They can then mark the active packets such that further negotiation in order to determine credentials is not needed by the "interior" switches. In essence, these "active firewalls" act as introducers of "outsiders" in a closed system.

- Naming services allow for UN-supervised but collision-free[3] (secure) identification of programs. The basis of this approach is to combine hashes of the code, public keys (and signatures), and user-defined strings to generate "names" for pieces of code. Thus, a certain program can have different names, each with different semantics and trust dependencies. For more details, see [AAKS98].

### 7.1.7 Conclusions and Future Work

This section has illustrated the security issues facing the architects of such systems. While the goal of Active Networks is to increase the set of design options available in distributed systems, the flexibility of a programmable infrastructure introduces considerable burdens for control, management, and security. The strategies for coping with the security aspects were presented in this section, however many issues currently remain unresolved, and are thus important directions for future work.

## 7.2 Secure DHCP

The DHCP protocol[Dro97] provides clients the ability to configure their networking and host specific parameters dynamically during the boot process. Eliminating the manual configuration each host at boot time. The typical parameters are the IP addresses of the client, gateways, and DNS server.

The technique described in the draft can provide both entity authentication and message authentication for DHCP. This network administrators may use this approach to protect against denial of service and resource misuse attacks. The approach is also helpful in preventing unintentionally

[3]To the extent that the cryptographic hash functions employed are resistant to collisions.

instantiated DHCP servers on the network.

The work described in this section is a combination of a proposal by Schiller-Huitema-Droms [Dro96] and work done by myself [DA98]. The draft is edited by myself and Ralph Droms of Bucknell University.

### 7.2.1 DHCP Threat Model

The threat to DHCP is inherently an insider threat (assuming a properly configured network where BOOTP ports are blocked on the enterprise's perimeter gateways.) Regardless of the gateway configuration, however, the insider and outsider threats are the same.

The threat specific to a DHCP client is the possibility of the establishment of a ”rogue” server with the intent of providing incorrect configuration information to the client. The motivation for doing so may be to establish a ”man in the middle” attack or it may be for a ”denial of service” attack.

There is another threat to DHCP clients from mistakenly or accidentally configured DHCP servers that answer DHCP client requests with unintentionally incorrect configuration parameters.

The threat specific to a DHCP server is an invalid client masquerading as a valid client. The motivation for this may be for ”theft of service”, or to circumvent auditing for any number of nefarious purposes.

The threat common to both the client and the server is the resource ”denial of service” attack. These attacks typically involve the exhaustion of valid addresses, or the exhaustion of CPU or network bandwidth.

### 7.2.2 Design Goals

These are the goals that were used in the development of the authentication protocol, listed in order of importance:

1. Address the threats presented above.
2. Avoid changing the current protocol.
3. Limit state required by the server.
4. Limit complexity (complexity breads design and implementation errors).

### 7.2.3 Approach

The approach is to provide a flexible mechanism for DHCP message authentication. Two methods for authentication are described. The first uses a shared password between two entities, and the second uses message authentication codes (MAC).

### 7.2.4 Status

The draft for the authentication of DHCP messages is currently in the DHCP working group. It is expected to move to working group last call before the next meeting (July 1999 in Oslo, Norway).

## 7.3 Secure-Identity Based Loading

As telecommuting and similar uses of computers and networks proliferate, a number of new security challenges arise. These result from a desire to graft the home environment onto the employer's network cloud, so that the home/office distinction is mainly geographic. The employee wishes to have a single resource model, similar in both environments. The employer wishes to preserve the security and integrity policies present in the office environment. If the employer believed that the home computer and network link were trustworthy, it might extend the office environment. A major challenge, then, is transparently establishing and maintaining the trustworthiness of home machines. We consider the case where the home user may also use the machine for personal tasks such as accounting, investment management, or allowing children to surf the Internet. In this common scenario, a single machine plays multiple roles.

The cost advantages, and convenience of telecommuting have made the technologically sophisticated home office a common phenomenon; it seems likely that it will become more common in the future. A major challenge which arises in this context is extending the office environment from work to home. Many companies have developed enterprise networking solutions that are very integrated and presume a standardized desktop, *e.g.*, one with certain applications available so that memoranda, spreadsheets, etc., can easily be shared amongst users on a LAN or secured intranet. The applications are often characterized by interactive graphical user interfaces and client/server organizations which require an environment very similar to, if not an exact replica of, the office environment, excepting the LAN/intranet connectivity. This connectivity is, if done properly, attractive to both employee and employer. (While it is not easy to do it properly, we demonstrate one powerful method in this paper.) The employee, ideally, when performing work for the employer while at home, would like the home environment to appear to be a remote desktop grafted on to the employer's network cloud, with whatever security policies required for that cloud. This grafting provides a useful application environment to the employee, *e.g.*, the facility to share documents, financial information in spreadsheet form, etc. The advantage of this approach is that an effective office environment remains effective when the user is at home, or even while traveling if a mobile office is required. The risk, of course, is that since the home or mobile environment is not controlled by the enterprise, the security and integrity constraints required for data sharing within the organization might not be enforced in the uncontrolled home environment. This is especially true when the required enforcement mechanisms are burdensome to the user. The employer wishes to preserve transparently the security and integrity policies present in the office environment. If the employer believed that the home computer and network link were trusted, it might extend the office environment. The user may also wish to use the machine for untrusted applications and personal tasks such as home accounting, investment management, browsing or recreation, or sharing the machine with children. This, then is the basic problem we address in this paper: transparently establishing the trustworthiness of home machines, and further their interconnection to the enterprise through an Internet. We view each use of the computer as one of multiple roles played by a single machine. We define the term role to mean the required external behavior of a system, i.e., the part that the system must play to participate in the enterprise network. The internetwork must be configured to support such roles, which may require route or packet discrimination, or the application of cryptographic transformations of the data to preserve privacy or verify integrity. The key goal for any design that supports multiple roles is that for roles with assumptions about the configuration of the machine, *e.g.*, the trusted roles implied by our earlier discussion of enterprise networks, the configuration must in fact be what is expected. We call such a configuration a personality; thus a particular personality is required for a given role. Examples of such configuration information include the presence of applications or device drivers, or even secrets required for inter-operation with the enterprise network (*e.g.*, by *grafting* the home network onto the enterprise network for the duration of work performed on behalf of the enterprise).

Secure Identity-Based Loading (SIBL) provides multiple personalities by partitioning the hard drive into n+1 partitions, where n is the number of supported personalities. One partition, the system partition, is used by all personalities for the core operating system components and shared applications. Each remaining partition is encrypted using a symmetric algorithm and associated with one and only one personality. During system initialization, the user identifies a role by submitting an extended public key certificate containing the desired attributes for the role's personality. The initialization process passes this role to the operating system after verifying that the user holds the private key associated with the submitted certificate. Upon start up, the operating system establishes the cryptographic context for decrypting the appropriate encrypted partition and instantiates the roles personality based on the attributes contained in the extended certificate. An overview of this process is shown in Figure 7.3.

### 7.3.1 SIBL Workstation

This section first describes an integrity policy for SIBL. It is followed by a presentation of the technical details required to implement SIBL using AEGIS as a basis. We then describe how and why smart cards are integrated. We finish the section by describing the changes required at the operating system level.

#### 7.3.1.1 Integrity Policy

SIBL's integrity policy is simple. The objects are the role and system partitions, and the bootstrap and network components. The only subject is the active role (or processes acting on its behalf). There are two classes of subjects. The first is the supervisor class. This is the class to which the system administrator belongs. Ideally for security reasons, there is only one member of this class. The second class is the normal class. The remaining roles belong to this class. A normal user

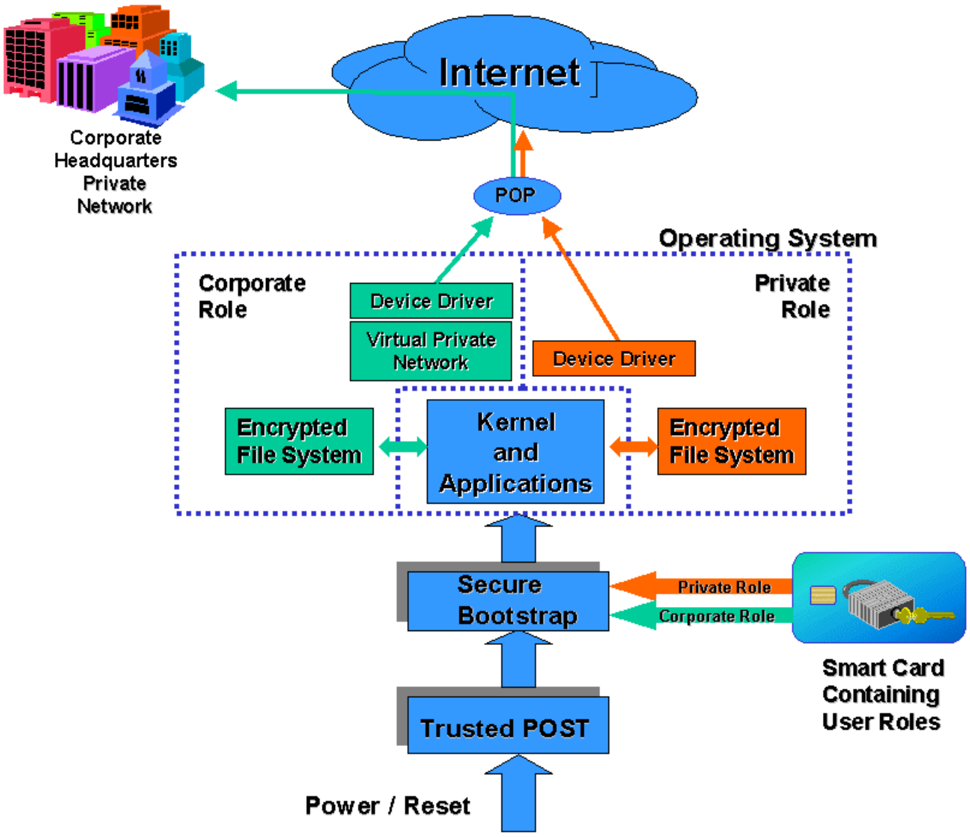


Figure 7.3: SIBL Overview

can only write to the partition associated with the active role, whereas a supervisor can write to any partition, including the bootstrap components, provided they have the decryption key for the partition. Both a normal and supervisor user can read the bootstrap components, and the system and active partitions. A supervisor user can also read and write any partition for which he has the correct key.

#### 7.3.1.2 Smart Card Integration

The use of smart cards eliminates the numerous problems introduced when having to store sensitive cryptographic material, *e.g.*, the role's private and partition keys, on the host system. Use of the smart card for secure storage implies, but does not require, using a single smart card for each role. We pass and authenticate the role's extended public key certificate in the bootstrap process, because we may wish to disable devices based on the expansion ROM scan performed during the bootstrap. Integrating the smart card at this level necessitates a reboot to change roles, but the need to clear memory and ensuring the integrity of the operating system also require a reboot to prevent potential covert channels.

#### 7.3.1.3 Operating System Modifications

Two changes must be made to the operating system kernel in order to support SIBL. First, the kernel must accept a role, identified by its extended certificate. Once the kernel has initialized itself, it must establish a connection with the users smart card to obtain a copy of the roles encryption key. Once the OS has the appropriate encryption key, it decrypts the files to configure the device drivers required by the current role and verifies their integrity. After the drivers are verified they are installed dynamically, and the appropriate network stacks layered on top of them thereby establishing the personality. Second, additional efforts must be made to ensure the integrity of the system partition. This can be done through the addition of an immutable file system, or best done with additional hardware protection.

#### 7.3.1.4 Network Integration

For purely local computing tasks, reliable control of the local workstation software is probably sufficient to support multiple, compartmentalized usage modes, conceived metaphorically as independent computing roles. However, insofar as corporate computing infrastructures have embraced client-server paradigms, computational integrity can depend more often on network environment than on individual workstation personality. Thus, practical use of multiple personality workstations also requires reliable control over the network environment in which the workstation operates. Integration of the workstation into a particular network environment must accomplish three goals:

1. Multiple workstation personalities must be protected one from another with respect to their use of local workstation resources.

2. Each workstation personality must be protected from integration with inappropriate network environments.

3. Each independent network environment must be protected from incompatible workstation personalities that may be latent within a single workstation.

While previous sections describe reliable construction and isolation of multiple, independent personalities within a single workstation (thereby addressing the first goal), the following sections apply extensions of the same basic mechanisms to reliably bind a particular personality to an appropriate network environment (thereby addressing the latter two goals).

## 7.4 Future Work - Dynamic Integrity

Now that the base of a computer system is secured, continuing and extending past work can move forward. Robust dynamic integrity protection mechanisms will for the first time be capable of providing strong intellectual property protection, as well protection for "dynamic" security systems such as intrusion detection monitors.

Solving this problem is difficult. However, a possible solution will likely involve controlling the memory management unit of the processor. This could be done with a software "wrapper" around access to the MMU, or possibly a redesign of the MMU itself. Both approaches, however, require more investigation.

# Chapter 8

# Conclusions

There are two principal contributions of this dissertation:

- A model for composing system layers into a high integrity system.
- The first complete implementation of a secure bootstrap process for the IBM personal computer.

## 8.1 CLIC Model

System designers can now use the CLIC model to assist them in designing robust high integrity systems. The underlying theory behind CLIC has been proven, Chapter 4, and the set of assumptions required to build a system have been enumerated. This provides for the first time a map for system designers in designing systems which require strong integrity guarantees.

## 8.2 AEGIS

AEGIS, Chapter 6, is the first embodiment of CLIC, and the first secure bootstrap for the IBM personal computer. AEGIS demonstrates two significant achievements:

- Demonstrates the practicality of CLIC, and
- significantly improves the reliability and security of the IBM personal computer system.

Current operating systems cannot provide security assurances since they are started via an untrusted process. With the explosive growth in Internet commerce, the need for security assurances from computer systems has grown considerably. AEGIS is a *guaranteed secure* boot process that ensures that the computer system is started via a trusted process, and ensures that the system starts in spite of integrity failures.

I believe that this work has a significant impact on the administration and manage-ability of systems. In this dissertation, I have demonstrated the need and provided an architecture for a secure bootstrap for any trusted system. I have also shown how that architecture can be utilized in a very realistic environment, with no loss of security. Thus, designers can build distributed computer systems of nodes which are in two logical states: (1) non-operational (e.g., down or recovering), and (2) operational and trusted. Such simple states and transitions ease, and in some sense make possible, verification of applications built on the distributed systems.

# Appendix A

## Initialization Code

This section includes the code that transitions the processor from real mode into protected mode. The code saves the current state of the processor, and calls the AEGIS main function.

```
;;; ----------------------------------------------------------------------
;;;
;;; $Copyright$
;;;
;;; $Id: start.tex,v 1.2 1999/03/08 03:22:13 waa Exp /03/08 00:34:53 waa Exp $
;;;
;;; The routines in this file move the proc from real mode (8086 emulation)
;;; to protected mode (32 bit 4GB address space) and back again. The code MUST
;;; be loaded at the address specified by RELOC.
;;;
;;;

%include "reloc.inc"

;;;
;;; ----------------------------------------------------------------------
;;; PUBLIC and EXTERNAL Symbols
;;;
;;;
EXTERN rtl_main
EXTERN Enable_gateA20

;;
;; Routines to map the E000 segment on the ROM into memory
;;
EXTERN SET_ROMCS_ON
EXTERN SET_ROMCS_OFF

;;;
;;; ----------------------------------------------------------------------

SEGMENT _TEXT PUBLIC CLASS=CODE USE32

        ;; Since we start out in real mode.
        BITS 16

;;; ----------------------------------------------------------------------
;;;
;;; This routine performs a mode switch from real mode
;;; to protected mode. This is the only entry point for
;;; both the BIOS and the RECOVERY code.
;;;

real_to_prot:
        ;; Set DS to CS since I'm using the CS as a DS.
        ;; Restore ax since we haven't saved state yet-
```

```
        ;; This is overkill no doubt.
        push ds
        push ax
        mov ax, SRELOC
        mov ds, ax
        pop ax

%ifdef BIOS
        ;; BX contains the value of "function" that was called
        ;; via the ISR. We store this value so we know which
        ;; mode to return the processor to the caller. It
        ;; will either be REAL or BIG REAL (4 GB Segs).
        ;;
        mov [_caller], bl

        ;; Save the registers that the calling functions
        ;; use to pass arguments to main. We save duplicates
        ;; to save the ops to mask out later.
        mov [_RegEAX], eax
        mov [_RegEBX], ebx
        mov [_RegECX], ecx
        mov [_RegEDX], edx
        mov [_RegCX], cx
        mov [_RegESI], esi
        mov [_RegEDI], edi
        mov [_RegSI], si
        mov [_RegES], es
        mov [_RegEDX], edx
        mov [_RegEBP], ebp

%ifdef TIME
        rdtsc
        ;; Save the timer values
        mov [starteax], eax
        mov [startedx], edx

        ;; Reset state. Use moves since stack is still 16bit
        mov edx, [_RegEDX]
        mov eax, [_RegEAX]
%endif

        ;; Save the real mode state here if you
        ;; need to transition back.
        ;; We don't need to save flags since an IRET is
        ;; done to return to the caller of the ISR.
        ;;
```

```
        ;; pusha is overkill on saving state
        ;;
        pusha
        push gs
        push fs
        push es

        ;; Now enable the E000 segment so we can find our
        ;; signatures/certificates
        push ds
        mov ax, 40h
        mov ds, ax
        call SET_ROMCS_ON
        pop ds

        ;; Now save the real mode stack location (SS:SP)
        ;;
        mov [real_mode_ss], ss
        mov [real_mode_sp], sp
%endif

        ;
        ; Set up a temp stack - This has to be here otherwise LGDT causes
        ; a real mode exception because the gdtarg is not in the range of
        ; of the SS Limit.
        ;
        ; THIS IS A PROBLEM IF ROM SIZE EVER EXCEEDS 64KB.
        ;
        mov bx, SRELOC ; new ss
        mov cx, 0xFF10 ; new sp
        mov ss, bx
        mov sp, cx

        ;; Now enable A20 for the extra address space
        call Enable_gateA20

        ;; Set up the gdt tables
        lgdt [gdtarg]

        ;; Switch into protected mode
        mov eax, cr0
        or eax, 1
        mov cr0, eax

        ;; Now do our far jump to our 32 bit code
        jmp dword CODESEG:start32
```

```
;;; ------------------------------------------------------------
;;;
;;; Start using 32 bit mode
        BITS 32
;;;
start32:

        ;; Now jump into the MSVC setup function from
        ;; Beacon located in mn0400.asm. It establishes the
        ;; environment and eventually calls _main.
        ;;
        jmp rtl_main

        ;; Exit point: jmp'd to from exit code in rtl_main
        ;; For NIC ROM do a reboot
        ;; For BIOS return to caller

global prot_to_real
prot_to_real:
        ;; First determine what state we need to return the processor
        ;; to the caller.
        ;;
        mov bl, [_caller]
        ;; If BL=3 then we return to a complete real mode state.
        ;; If BL is anything else, then we return with ES and
        ;; DS limits set to 4GB.
        cmp bl, 0x3
        jz restore_real
        mov dx, BIGREALSEG
        mov ds, dx
        mov es, dx
        mov dx, REALDATASEG
        jmp skip_ds_es

restore_real:
        ;; Set the stack and other segments back to 16 bit
        ;; with 64KB limits.
        mov dx, REALDATASEG
        mov ds, dx
        mov es, dx
skip_ds_es:
        mov ss, dx
        mov fs, dx
        mov gs, dx
```

```
        ;; Clear stack pointer in prep of resetting it.
        xor esp, esp

        ;; Go back to real mode
        ;; First set the cs selector to a limit of 64KB via
        ;; a jump.
        o16
        ; For some reason NASM or the linker produces incorrect
        ; machine instructions for this statement. I suspect the linker.
        ; Hand code it.
; jmp REALSEG:start16
        db 0xea
        dw start16
        dw REALSEG

        BITS 16
start16:
        ;; Now set the PE bit
        mov eax, cr0
        and al, 0xfe
        mov cr0, eax

        ;; Now jump to clear the cache and enable real mode again
        o16
        ; For some reason NASM or the linker produces incorrect
        ; machine instructions for this statement. I suspect the linker.
        ; Hand code it.
; jmp SRELOC:start16
        db 0xea
        dw realmode
        dw SRELOC

realmode:
%ifdef BIOS
        ;; Restore the state of the processor. First restore the real
        ;; mode stack.
        mov ss, [real_mode_ss]
        mov sp, [real_mode_sp]

        ;; Now turn off the E000 segment mapping
        push ds
        mov ax, 40h
        mov ds, ax
        call SET_ROMCS_OFF
        pop ds
```

```
        ;; Now pop off the processor state from the old stack.
        ;; XXXXX TO DO XXXXXX
        ;; Need to do something to clear the high word in these
        ;; registers. It doesn't cause a problem, but it may
        ;; with another bios.
        pop es
        pop fs
        pop gs
%ifdef TIME
global Foo
Foo:
        rdtsc
        ;; Set the data segment so we can get to the stored vals
        mov bx, SRELOC
        mov ds, bx
        sub eax, [starteax]
        sbb edx, [startedx]
        ;; Make a global symbol so we can stop and read the value
global TimerStop
TimerStop:
%endif
        popa
        pop ds

        ;; Return to the ISR
        iret

;;; ----------------------------------------------------
;;; ----------------------------------------------------
;;;
;;; INPUT: Nothing
;;;
;;; OUTPUT: Nothing - machine in real mode
;;;
;;; USES: eax, dx
;;;
        BITS 32

        global ProtToReal
ProtToReal:
        pop eax
        sub eax, RELOC
        push ax
        sub esp, RELOC ;; fixup the stack

        ;; Now set up the segment descriptors
```

```
            mov dx, REALDATASEG ;; setup data descriptor
            mov ds, dx
            mov ss, dx
            mov es, dx

            o16
; jmp REALSEG:begin16 ;; setup code descriptor
            db 0xea ;; via a far jump
            dw begin16
            dw REALSEG

            BITS 16
begin16:
            mov eax, cr0 ;; set CR0
            and al, 0xfe
            mov cr0, eax
            o16
; jmp SRELOC:intrealmode ;; Jump to reload CS
            db 0xea
            dw intrealmode
            dw SRELOC

intrealmode: ;; Fully in real mode now.
            sti ;; Enable interrupts
            mov ax, cs ;; set the segment offsets
            mov ds, ax
            mov ss, ax
            mov es, ax
            ret

;;; ----------------------------------------
;;; ----------------------------------------
;;;
            global RealToProt
RealToProt:
            cli ;; disable interrupts
            lgdt [gdtarg] ;; Set up the gdt tables
            mov eax, cr0 ;; Switch into protected mode
            or eax, 1
            mov cr0, eax
            jmp dword CODESEG:restart32 ;; far jump to 32 bit code
                                                ;; to set the cs selector.

;;; ----------------------------------------
;;;
;;; Start using 32 bit mode
```

```
BITS 32

restart32:
        ;; Set the segment selectors
        mov ax, DATASEG
        mov ds, ax
        mov ss, ax
        mov es, ax

        add esp, RELOC ;; Fix the stack
        xor eax, eax ;; Fix up the return address
        pop ax ;; pop the 16 bit return
        add eax, RELOC ;; add the shifted segment
        push eax ;; push the 32bit addr
        ret

;;; -----------------------------------------------------------
;;;
        BITS 16

real_mode_ss: dw 0
real_mode_sp: dw 0
starteax: dd 0
startedx: dd 0

;; Make caller global so we can access it in _main
;; and call the correct function.
global _caller
_caller: db 0

;; Variables to store register values for the main
;; function.
global _RegEAX
_RegEAX dd 0

global _RegEBX
_RegEBX dd 0

global _RegECX
_RegECX dd 0

global _RegEDX
_RegEDX dd 0

global _RegCX
_RegCX dw 0
```

```
global _RegESI
_RegESI dd 0

global _RegEDI
_RegEDI dd 0

global _RegEBP
_RegEBP dd 0

global _RegSI
_RegSI dw 0

global _RegES
_RegES dw 0

%else ;; RECOVERY
        ;; Reboot and try again. Assume at least a 486.. . . no need to
        ;; do the triple-fault

%endif


;;; -----------------------------------------------------------
;;;
;;; DUMB GDT TABLE
;;;
gdt_table:
        dd 0,0 ; NULL Descriptor

        ;; Prot Mode CODE SEG
        ;
        dw 0xFFFF ;4.2GB
        dw 0 ;Base of 00000000h
        db 0, 0x9A ;Code segment, rx access
        db 0xCF, 0 ;32-bit segment

        ;; Prot Mode DATA SEG
        ;
        dw 0xFFFF ;4.2GB
        dw 0 ;Base of 00000000h
        db 0,0x92 ;Data segment, rw
        db 0xCF, 0 ;32-bit segment

        ;; REAL Mode Code SEG
        ;
```

```
        dw 0xFFFF ; segment limit 00 .. 15
        dw 0 ; A00 .. A15 base address
        db 0x6 ; A16 .. A23 base address (0x60000)
        db 0x9b ; segment access rights
        db 0x0 ; granularity, op-size, Limit A16 .. A19
        db 0x0 ; A24 .. A31 base address

        ;; REAL Mode Data SEG
        ;
        dw 0xFFFF ; segment limit 00 .. 15
        dw 0 ; A00 .. A15 base address
        db 0x6 ; A16 .. A23 base address (0x60000)
        db 0x93 ; segment access rights
        db 0x0 ; granularity, op-size, Limit A16 .. A19
        db 0x0 ; A24 .. A31 base address

        ;; Big REAL Mode SEG
        ;
        dw 0xFFFF ; segment limit 00 .. 15
        dw 0 ; A00 .. A15 base address
        db 0x6 ; A16 .. A23 base address
        db 0x93 ; segment access rights
        db 0x8F ; granularity, op-size, Limit A16 .. A19
        db 0x0 ; A24 .. A31 base address

                        ; Computing the length automatically
                        ; works inconsistently due to problems
gdtarg: ; with either the linker or NASM.
        dw 0x30 ; ; limit
        dd gdt_table ; base
```

# Appendix B

## Common Code

```
/******************************************************************
 * *
 * $Copyright$
 * *
 * Filename: cert.c *
 * *
 * This file contains the core certificate routines. *
 * *
 ******************************************************************/

#include "icert.h"

extern int Verify(unsigned char *, unsigned long, unsigned char *,
                  unsigned int);

extern unsigned short CurrentYear(void);

extern unsigned char CurrentMonth(void);

extern unsigned char CurrentDay(void);


// This procedure determines the validity of a given
// certificate. It assumes that expiration of the
// cert is midnight of the certificates expir date.
// The procedure returns 0 if valid.
//
// The pointer is passed in as an unsigned char because
// we don't want the calling programs to know the
// structure of the cert in case we want to change it
// later.
//
int
VerifyCert (unsigned char *pucCert)
{
        struct cert *pCert;
        int status;
        unsigned short usYear;
        unsigned char ucMonth;
        unsigned short tmp;
        unsigned char tmp1;

        pCert = (struct cert *)pucCert;
        tmp1 = pCert->ucType;
        tmp = pCert->usVendor;
        tmp = pCert->usDevice;
```

```
        tmp1 = pCert→ucRev;
        tmp1 = pCert→ucPclass;
        tmp1 = pCert→ucSclass;
        tmp1 = pCert→ucBclass;
        tmp1 = pCert→ucMonth;
        tmp1 = pCert→ucDay;
        tmp = pCert→usYear;

        usYear = CurrentYear();
        ucMonth = CurrentMonth();
        // Check for expiration of the Cert
        if (usYear == pCert→usYear) {
          // Check the month
          if (ucMonth == pCert→ucMonth) {
            // Check day
            if (CurrentDay() > pCert→ucDay) {
              status = CERTEXPIRED;
              goto Error;
            }
          } else if (ucMonth > pCert→ucMonth) {
            status = CERTEXPIRED;
            goto Error;
          }
        } else if (usYear > pCert→usYear) {
          status = CERTEXPIRED;
          goto Error;
        }

        // Verify the Certificate
        status = Verify(pucCert, sizeof(struct cert) - SIGLEN,
                        pCert→pucSig, SIGLEN);

Error:
        return status;
}

unsigned char *
GetDeviceHash(unsigned char *pucCert)
{
  struct cert *pCert;

  pCert = (struct cert *)pucCert;

  return pCert→pucDevHash;
}
```

```
unsigned long
GetVendorDevice(unsigned char *pucCert)
{
   unsigned long *pulTmp;
   unsigned long ulTmp;

   // Stupid pointer tricks
   pulTmp = (unsigned long *)&(((struct cert *)pucCert)->usVendor);
   ulTmp = *pulTmp;

   return ulTmp;
}

unsigned long
GetBases(unsigned char *pucCert)
{
   unsigned short *pusTmp;
   unsigned short usTmp;
   unsigned long ulTmp;

   // unfortunately - I screwed up and misordered the bases in
   // the cert structure. I'm paying for it now.
   pusTmp = (unsigned short *)&(((struct cert *)pucCert)->ucSclass);
   usTmp = *pusTmp;
   ulTmp = (unsigned long)((usTmp << 8) | ((struct cert *)pucCert)->ucRev);
   ulTmp = (ulTmp << 8) | ((struct cert *)pucCert)->ucPclass;

   return ulTmp;
}
```

# Appendix C

## AEGIS

```
/************************************************************
 * *
 * $Copyright$ *
 * *
 * *
 * This file is the main verification function. It is named *
 * main primarily so we can link everything together without *
 * the compilers/linkers complaining. *
 * *
 ************************************************************/

#include <string.h>
#include "recov.h"
#include "verify.h"

void
main (void)
{
  unsigned char *pucTargAddr;

  /* The external variable caller is an unsigned char indicating
      the routine that kicked this whole thing off. It is set in
      start.asn.
  */
 switch (caller) {

    /* VERIFYBIOS - verify both the F000 and E000 segments. */
 case VERIFYBIOS:
    if (VerifyBIOS()) {
       /* Signature Mismatch */
       Dum_Beep(0x5);
    }
    break;

    /* Verify a PnP (ISA) ROM. This routine uses RegES and
       RegSI. RegES:RegSI is the address of the ROM.
    */
 case VERIFYPnP:
    pucTargAddr = (unsigned char *)Make32Addr(RegES, (unsigned long)RegSI);
    if (VerifyPnPROM(pucTargAddr)) {
       Dum_Beep(0xF);
    }
    break;

    /* Verify a PCI ROM. RegES:RegESI is the address of the ROM.
       RegCX is the number of pages of the ROM. RegEAX contains the
```

```
        vendor_id and the device_id. RegEDX contains the various classes.
    */
  case VERIFYPCI:
    pucTargAddr = (unsigned char *)RegESI;
    if (VerifyPCI(pucTargAddr, RegCX, RegEAX, RegEDX)) {
      Dum_Beep(0xF);
    }

    break;

    /* Verify the boot sector. RegES:RegSI is the address of the
     * boot sector. RegCX is the number of bytes in 512 byte sectors.
     * RegEAX contains the signature location.
     */
  case VERIFYBBLK:
    pucTargAddr = (unsigned char *)Make32Addr(RegES, (unsigned long)RegSI);
    if (VerifyBootBlk(pucTargAddr, RegCX, (unsigned char *)RegEAX)) {
      Dum_Beep(0x5);
      printf("Invalid boot block found.  Attempting recovery ...\r\n");
      RecoverBootBlk(pucTargAddr, RegCX, (unsigned char *)RegEAX);
      Dum_Beep(0xF);
      /* ClearScreen(); */
    }
    break;

    /* Verify the kernel's header, data, and text areas.
     *
     * int VerifyKern(iHdrAddr, iHdrSize, iTextAddr, iTextAddr,
     * iDataAddr, iDataSize)
     */
  case VERIFYKERN:
    if (VerifyKern(RegEAX, RegEBP, RegECX, RegEDX, RegESI, RegEDI)) {
      Dum_Beep(0xF);
    }
    break;
  }

  /* Any return value is ignored. Any problems with the verification
      are handled within this function, e.g. we call the RECOVERY ROM
      from this routine.
  */
}
```

```
/************************************************************
 * *
 * $Copyright$ *
 * *
 * *
 * This file contains the recovery code. Depending on what *
 * needs recovered- it is either handled here or jumps to the *
 * NIC card ROM. *
 * *
 ************************************************************/

#include "recov.h"

/*
 * Recover the primary boot block. Right now we only recover a
 * PicoBSD primary sector. Anything else is skipped.
 */

void
RecoverBootBlk (unsigned char *pucBlkAddr, unsigned short usSize,
                unsigned char *pucSigAddr)
{
  unsigned long *pulTmp;

  /* Skip over MSDOS disks for now by looking for FAT12 at offset
   * 0x7C36. This seems to be the only common element of MS boot
   * sectors.
   */
  pulTmp = 0x7C36;
  if (*pulTmp == 0x31544146) {
    printf("MSDOS disks are not supported yet.\r\n");
    return;
  }

  /*
   * Anything else is a PicoBSD boot sector. Copy a valid one
   * over from data space for now. Don't try and fix it on disk
   * since that requires a protected to real mode transition.
   *
   */
  if (usSize == 1) {
    printf("Fixing PicoBSD primary boot sector ...\r\n");
    if (FixBoot1(&ucPicoBoot1) ≠ 0) {
      /* There was an error fixing the disk */
      printf("ERROR Fixing primary sector.  Spinning!\r\n");
      printf("You'll have to manually reset!\r\n");
```

```
      while (1)
        ;
    }
  }

  printf("Recovery complete.  Continuing bootstrap.\r\n");

  return;
}
```

# Appendix D

## Boot Loader Code

# Appendix E

## Administrative Programs

```
/* $Copyright$ */

/*********************************************************************
 * *
 * This program scans the memory range 0xC0000 through 0xF6000 *
 * Looking for valid ISA ROM images. A valid ROM image is identified *
 * by the following memory structure: *
 * DW 0xAA55 ; Signature Word (Stored Little-Endian) *
 * DB len ; length in 512-byte blocks (0=256) *
 * ..code bytes.. *
 * DB checksum; Twos-complement of all preceeding *
 * ; bytes from start. This byte MUST *
 * ; be the last byte in the image *
 * ; computed from the "len" byte. *
 * NOTE: If all (len * 512) bytes are summed in a modulo-256 fashion, *
 * the sum byte will be zero (00H), since the checksum byte is *
 * the twos-complement of all preceeding values, and will result *
 * in zero when added to the computed value. *
 * *
 * This program is designed to be used with Watcom C and linked with *
 * a DOS extender. *
 * *
 *********************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "crypto.h"

#define START_ADDR 0xC0000                    /* Scan starting address */
#define END_ADDR 0xF6000                      /* Scan ending address */


/* Compute the simple additive ROM checksum. Return the
 * result. A valid checksum is zero.
 */
unsigned char
checksum(unsigned char *pMem, unsigned int uiSizeInBytes)
{
  unsigned char ucSum = 0;
  unsigned int i;

  for (i = 0; i < uiSizeInBytes; i++) {
    ucSum += *pMem;
    pMem++;
```

```
    }

    return ucSum;
}

/* Main program. Does not take any arguments. */
int
main (int argc, char *argv[])
{
    unsigned char *pMemAddr;
    unsigned char *pRomEnd;
    unsigned long ulRomSize;
    unsigned char ucRomSum;
    unsigned char ucSignature[MAX_SIGNATURE_LEN];
    unsigned char ucHdr[4];
    unsigned int uiSigLen = 0;
    char Filename[32];



    /* Start our memory scan in 2KB byte chunks.
     */
    printf ("Starting ISA ROM scan\n");

    pMemAddr = (unsigned char *)START_ADDR;

    while (pMemAddr <= (unsigned char *)END_ADDR) {
        if (*((unsigned short *)pMemAddr) == 0xAA55) {
            /* Grab the size of the image */
            ulRomSize = *(pMemAddr+2) * 512;
            pRomEnd = pMemAddr + ulRomSize;

            /* Let someone know we found something */
            printf("\tFound ROM at %X to %X\n", pMemAddr, pRomEnd);

            /* Now compute the additive checksum to ensure a valid ROM */
            ucRomSum = checksum(pMemAddr, ulRomSize);
            if (ucRomSum == 0) {
                printf ("\t\tROM checksum valid\n");
                // We have a valid ROM so create a signature for it.
                Sign(pMemAddr, ulRomSize, ucSignature, &uiSigLen);
                // Verify it to ensure there are any errors.
                if (Verify(pMemAddr, ulRomSize, ucSignature, uiSigLen) == 0){
                    printf("\t\tSignature Verified\n");
                    // Now write it out to disk
                    sprintf(Filename, "%05X.sig", pMemAddr);
```

```
                // Set the ID in the file. The ID for an ISA ROM is simply
                // its address in memory.
                memcpy(ucHdr, pMemAddr, 4);
                WriteSig(ucHdr, 4, ucSignature, uiSigLen, Filename);
                // Now be real paranoid and read it and verify
                // memset vars to zero first
                memset(ucHdr, 0, 4);
                memset(ucSignature, 0, MAX_SIGNATURE_LEN);
                ReadSig(ucHdr, 4, ucSignature, uiSigLen, Filename,
                        MAX_SIGNATURE_LEN);
                if (Verify(pMemAddr, ulRomSize, ucSignature, uiSigLen) == 0){
                  printf("\t\tSignature Verified from file\n");
                }
            }
        } else {
            printf ("\t\tROM checksum invalid:  %X\n", ucRomSum);
        }
    }
    // Increase offset by 2KB bytes and continue the scan
    pMemAddr += 0x200;
  }

  return 0;
}
```

# Bibliography


[AAH$^+$98] D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare Active Network Architecture. *IEEE Network Magazine*, 12(3):29–36, 1998.

[AAKS98] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A Secure Active Network Environment Architecture: Realization in SwitchWare. *IEEE Network Magazine*, 12(3):37–45, 1998.

[AD97] S. Alexander and R. Droms. DHCP Options and BOOTP Vendor Extensions. Internet RFC 2132, March 1997.

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[AKFS98] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated Recovery in a Secure Bootstrap Process. In *Proceedings of Network and Distributed System Security Symposium*, pages 155–167. Internet Society, March 1998.

[Ale98] D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, September 1998.

[AMP] Ashar Aziz, Tom Markson, and Hemma Prafullchandra. Assigned Numbers for SKIP Protocols. http://skip.incog.com/spec/numbers.html.

[Any95] Charles A. Anyimi. Implementing A Plug and Play BIOS Using Intel's Boot Block Flash Memory. Technical Report AP-608, Intel Corporation, February 1995.

[Arb94] William A. Arbaugh. Digitally Signed Execution. Presentation at the Office of INFOSEC Research Monthly Review, December 1994.

[Arb95] William A. Arbaugh. AEGIS: A High Integrity Computer System. Unpublished technical report, January 1995.

[Atk95] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.

[BFIK98a] M. Blaze, J. Feigenbaum, J. Ioannidis, and A.D. Keromytis. The KeyNote Trust Management System. Work in Progress, June 1998.

[BFIK98b] M. Blaze, J. Feigenbaum, J. Ioannidis, and A.D. Keromytis. The KeyNote Trust Management System. Work in Progress, June 1998.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *JACM*, 42(1):269–291, January 1995.

[Bon97] Vesselin Bontchev. Re: Flash virus. USENET message posted to comp.virus, June 1997.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Internet RFC 2208, 1997.

[CH94] Paul C. Clark and Lance J. Hoffman. BITS: A Smartcard Protected Operating System. *Communications of the ACM*, 37(11):67 – 70, November 1994.

[Cla94] Paul Christopher Clark. *BITS: A Smartcard Protected Operating System*. PhD thesis, George Washington University, 1994.

[CLC94] Compaq Computer Corporation, Phoenix Technologies Ltd., and Intel Corporation. Plug and Play BIOS Specification. Technical Report Version 1.0A, Intel, May 1994.

[Com89] Consultation Committee. *X.509: The Directory Authentication Framework*. International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[Cor85] Intel Corporation. *External Product Specification 386 Object Modules Format*, rev. b. x005 edition, September 1985.

[DA98] Ralph Droms and William Arbaugh. Authentication for dhcp messages. Work in Progress, November 1998.

[Dee89] S. E. Deering. Host extensions for IP multicasting. Internet RFC 1112, 1989.

[DH76] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, Nov 1976.

[DOD85] DOD. Trusted Computer System Evaluation Criteria. Technical Report DOD 5200.28-STD, Department of Defense, December 1985.

[dri96] daemon9, route, and infinity. Project neptune. *Phrack Magazine*, 7(48), 1996.

[Dro96] Ralph Droms. Authentication for dhcp messages. Work in Progress, March 1996.

[Dro97] R. Droms. Dynamic Host Configuration Protocol. Internet RFC 2131, March 1997.

[DvOW92] W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[EFRT97] Carl M. Ellison, Bill Frantz, Ron Rivest, and Brian M. Thomas. Simple Public Key Certificate. Work in Progress, April 1997.

[EKO94] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. The operating system kernel as a secure programmable machine. In *Proceedings of the Sixth SIGOPS European Workshop*, pages 62–67, September 1994.

[Eli96] Julian Elischer. 386 boot. /sys/i386/boot/biosboot/README.386, July 1996. 2.1.5 FreeBSD.

[Fit98] Nick FitzGerald. CIH Virus Information. http://www.virusbtn.com/VirusInformation/CIH.html, July 1998.

[GDM89] Y. Desmedt G. Davida and B. Matt. Defending Systems Against Viruses through Cryptographic Authentication. In *1989 IEEE Symposium on Security and Privacy*, pages 312–318. IEEE, 1989.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[Gon92] L. Gong. A Security Risk of Depending on Synchronized Clocks. *ACM Operating Systems Review*, 26(1), January 1992.

[Gri93] R. Grimes. AT386 Protected Mode Bootstrap Loader. /sys/i386/boot/biosboot/README.MACH, October 1993. 2.1.5 FreeBSD.

[GS95] Li Gong and Paul Syverson. Fail-Stop Protocols: An Approach to Designing Secure Protocols. In *Proceedings of IFIP DCCA-5*, September 1995.

[Haz95a] Peter Hazen. Intel's Flash Memory Boot Block Architecture for Safe Firmware Updates. Application Brief AB-57, Intel, December 1995.

[Haz95b] Peter Hazen. Intel's Flash Memory Boot Block Architecture for Safe Firmware Updates. Technical Report AB-57, Intel Corporation, December 1995.

[HB96] L.T. Heberlein and M. Bishop. Attack Class: Address Spoofing. In *Proceedings of the 19th National Information Systems Security Conference*, pages 371–377, October 1996.

[HKK93] Hermann Härtig, Oliver Kowalski, and Winfried Kühnhauser. The Birlix security architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[HKM$^+$98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical report, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[Jen93] William Jenkins. The RATBAG Report. Internal U.S. D.O.D. report, Fall 1993.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC:Keyed–Hashing for Message Authentication. Internet RFC 2104, February 1997.

[Kla97] Christopher Klaus. Backdoors. Unpublished paper, August 1997.

[KS93] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of TRIPWIRE: A File System Integrity Checker. Technical Report TR-93-071, Department of Computer Science, Purdue University, November 1993.

[KS94] G. Kiim and E. Spafford. Experience with tripwire: Using integrity checkers for intrusion detection. In *System Administration, Networking, and Security Conference III*. USENIX, 1994.

[LAB92] Butler Lampson, Martin Abadi, and Michael Burrows. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, v10:265–310, November 1992.

[Ler95] Xavier Leroy. *The Caml Special Light System (Release 1.10)*. INRIA, France, November 1995.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[LR97] B. Lampson and R. Rivest. SDSI - Simple Distributed System Infrastructure (Draft). Technical report, MIT, 1997.

[Ltd91] Phoenix Technologies Ltd. *System BIOS for IBM PCs, Compatibles, and EISA Computers*. Addison Wesley, 2nd edition, 1991.

[MBD89] F. Mayer M. Branstad, H. Tajalli and D. Dalva. Access mediation in a message-passing kernel. In *IEEE Conference on Security and Privacy*, pages 66–71, 1989.

[MH95] G. Malkin and A. Harkin. TFTP Option Extension. Internet RFC 1782, March 1995.

[Mic] Microsoft. Authenticode Techonology. Microsoft's Developer Network Library, October 1996.

[MMO+94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical report, Department of Computer Science, University of Arizona, June 1994.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NBS77] Data Encryption Standard, January 1977.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[Neu95] Peter G. Neumann. Architectures and Formal Representations for Secure Systems. Final Report. SRI Project 6401 A002, SRI International, October 1995.

[NL96a] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI '96)*, 1996.

[NL96b] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI '96)*, 1996.

[oS94] National Institute of Standards. Digital Signature Standard. Technical Report FIPS-186, U.S. Department of Commerce, May 1994.

[oS95] National Institute of Standards. Secure Hash Standard. Technical Report FIPS-180-1, U.S. Department of Commerce, April 1995. Also known as: 59 Fed Reg 35317 (1994).

[pan96] Cracker Attack Paralyzes PANIX. RISKS Digest. Volume 18. Issue 45., September 1996.

[Par74] David L. Parnas. On a "Buzzword": Hierarchical Structure. In *Information Processing 74*, pages 336–339. North-Holland, 1974.

[Par77] David L. Parnas. The Influence of Software Structure on Reliability. In *Current Trends in Programming Methodology*, pages 111–119. Prentice Hall, April 1977.

[Pen98a] Adam L. Penenberg. Hacking Bhabha. http://www.forbes.com/tool/html/98/Nov/1116/feat.html, November 1998.

[Pen98b] Adam L. Penenberg. Tracking the Hack. *Forbes Magazine*, 162(11):132–138, November 1998.

[PG89] Maria M. Pozzo and Terrence E. Gray. A Model for the Containment of Computer Viruses. In *1989 IEEE Symposium on Security and Privacy*, pages 312–318. IEEE, 1989.

[PJ96] C. Partridge and A. Jackson. Smart Packets. Technical report, BBN, 1996. http://www.net-tech.bbn.com-/smtpkts/smtpkts-index.html.

[Pos81] Jon Postel. Internet Protocol. Internet RFC 791, 1981.

[RL98] Luke Reiter and Jim Louderback. The Mother of All Viruses. http://www.zdnet.com/zdtv/cybercrime/features/story/0,3700,2121893,00.html, 1998.

[RP94] J. Reynolds and J. Postel. Assigned Numbers. Internet RFC 1700, October 1994.

[Sch75] M.D. Schroeder. Engineering a Security Kernel for MULTICS. In *Fifth Symposium on Operating Systems Principles*, pages 125–132, November 1975.

[Sch96] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 2 edition, 1996.

[Sku96] Fridrik Skulason. Re: Flash virus. USENET message posted to comp.virus, May 1996.

[Sol92] K. R. Sollins. The TFTP Protocol (revision 2). Internet RFC 1350, July 1992.

[SW96] Edward Solari and George Willse. *PCI Hardware and Software Architecture and Design*. Annabooks, third edition, 1996.

[Syv94] P. Syverson. A Taxonomy of Replay Attacks. In *Proceedings of the Computer Security Foundations Workshop VII (CSFW7)*, June 1994.

[TY91] J.D. Tygar and Bennet Yee. DYAD: A System for Using Physically Secure Coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. Ants: A toolkit for building and dynamically deploying network protocols. In *IEEE OpenArch Proceedings*. IEEE Computer Society Press, April 1998.

[Yee94] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[Zim95] P. Zimmerman. PGP User's Manual, 1995.