Automated Recovery in a Secure Bootstrap Process

William A. Arbaugh
Angelos D. Keromytis
David J. Farber*
Jonathan M. Smith
University of Pennsylvania
Distributed Systems Laboratory
Philadelphia, PA. 19104-6389
{waa, angelos, farber, jms}@dsl.cis.upenn.edu

November 21, 1997

Abstract

Integrity is rarely a valid presupposition in many systems architectures, yet it is necessary to make any security guarantees. To address this problem, we have designed a secure bootstrap process, AEGIS, which presumes a minimal amount of integrity, and which we have prototyped on the Intel x86 architecture. The basic principle is sequencing the bootstrap process as a chain of progressively higher levels of abstraction, and requiring each layer to check a digital signature of the next layer before control is passed to it. A major design decision is the consequence of a failed integrity check. A simplistic strategy is to simply halt the bootstrap process. However, as we show in this paper, the AEGIS bootstrap process can be augmented with automated recovery procedures which preserve the security properties of AEGIS under the additional assumption of the availability of a trusted repository. We describe two means by which such a repository can be implemented, and focus our attention on a network-accessible repository.

*Smith and Farber's work is supported by DARPA under Contracts #DABT63-95-C-0073, #N66001-96-C-852, and #MDA972-95-1-0013 with additional support from the Hewlett-Packard and Intel Corporations.

1 Introduction

Systems are organized as layered levels of abstraction, in effect defining a series of virtual machines. Each virtual machine presumes the correctness (*integrity*) of whatever virtual or real machines underlie its own operation. Without integrity, no system can be made secure, and conversely, any system is only as secure as the foundation upon which it is built. Thus, without such a secure bootstrap the operating system kernel cannot be trusted since it is invoked by an untrusted process. We believe that designing trusted systems by explicitly trusting the boot components provides a false sense of security to the users of the operating system, and more important, is unnecessary.

We have previously reported[AFS97] the design and preliminary implementation results for AEGIS, a secure bootstrap process. AEGIS increases the security of the boot process by ensuring the integrity of bootstrap code. It does this by constructing a chain of integrity checks, beginning at power-on and continuing until the final transfer of control from the bootstrap components to the operating system itself. The integrity checks compare a computed cryptographic hash value with a stored digital signature associated with each component.

The importance of the integrity of the bootstrap process is highlighted by the recent disclosure by Intel that the Pentium Pro and Pentium II processors can have their microcode dynamically updated through a process during bootstrap by using capabilities of the Basic Input Output System (BIOS) and the Power on Self Test (POST)[Gol97]. While this is a useful upgrade capability, it is also a dangerous one. AEGIS provides integrity guarantees not only for the BIOS code that updates the processor microcode, but also the microcode itself. Beyond the integrity guarantees, AEGIS can also provide secure automatic updates to all of the bootstrap components including the processor microcode.

In order to provide integrity guarantees the AEGIS model relies explicitly on three assumptions:

1. The motherboard, processor, and a portion of the system ROM (BIOS) are not compromised, *i.e.*, the adversary is unable or unwilling to replace the motherboard or BIOS.

2. Existence of a cryptographic certificate authority infrastructure to bind an identity with a public key, although no limits are placed on the type of infrastructure.

3. A trusted repository exists for recovery purposes. This repository may be a host on a network that is reachable through a secure communications protocol, or it may be a trusted ROM card located on the protected host.

The AEGIS architecture, which we outline below in Section 2, includes a recovery mechanism for repairing integrity failures and protection against some classes of denial of service attacks. An added benefit of the recovery mechanism is the potential for reducing the Total Cost of Operation (TCO) of a computer system by reducing trouble calls and down time associated with failures and upgrades of the boot process.

From the start, AEGIS has been targeted for commercial operating systems on commodity hardware, making it a practical "real-world" system. In AEGIS, the boot process is guaranteed to end up in a secure state, even in the event of integrity failures outside of a minimal section of trusted code.

We define a *guaranteed secure* boot process in two parts. The first is that no code is executed unless it is either explicitly *trusted* or its integrity is verified prior to its use. The second is that when an integrity failure is detected a process can recover a suitable verified replacement module. This recovery process is the focus of the current paper.

1.1 Responses to integrity failure

When a system detects an integrity failure, 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 component.

The second is to not use or execute the component. 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 component.

The first two approaches are unacceptable when the systems are important network elements such as switches, intrusion detection monitors, or associated with electronic commerce, since they either make the component unavailable for service, or its results untrustworthy.

1.2 Goals

There are six main goals of the AEGIS recovery protocol.

1. Allow the AEGIS client and the trusted repository to mutually authenticate their identities with limited or no prior contact (mobility between domains).

2. Prevent man in the middle attacks.

3. Prevent replay attacks.

4. Mitigate certain classes of denial of service attacks.

5. Allow the participating parties to agree upon a shared secret in a secure manner in order to optimize future message authentication.

6. KISS (Keep It Simple and Secure): Complexity breeds design and implementation vulnerabilities.

1.3 Outline of the Paper

In Section 2, we make the goals of the AEGIS design explicit. Sections 3, 4, and 5 form the core of the paper, giving an overview of AEGIS, and the IBM PC boot process. Section 4 provides an introduction to the cryptographic and system tools needed to build a secure recovery protocol, and describes such a protocol. Section 5 describes the details of adding the recovery protocol to existing Dynamic Host Configuration Protocol (DHCP), and Trivial File Transfer Protocol (TFTP) implementations and provides performance information. We discuss the system status and our next steps in section 6, and conclude the paper in section 7.

2 AEGIS Architecture

2.1 Overview

To have a practical impact, AEGIS must be able to work with commodity hardware with minimal changes (ideally none) to the existing architecture. The IBM PC architecture was selected as our prototype platform because of its large user community and the availability of the source code for several operating systems. We also use the FreeBSD operating system, but the AEGIS architecture is not limited to any specific operating system. Porting to a new operating system only requires a few minor changes to the boot sector code so that the kernel can be verified prior to passing control to it. Since the verification code is contained in the BIOS, the changes will not substantially increase the size of the boot loader, nor the boot sector.

AEGIS modifies the standard PC boot process shown in Figure 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 and additions to the BIOS. The BIOS contains the verification code, and public key certificate(s). In essence, the 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]. 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 "warm booted" to ensure that the system starts. This entire process occurs without user intervention.

In addition to ensuring that the system starts 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 signature for each expansion card[1], any additional expansion cards will fail the integrity test. Similarly, a new operating system cannot be started since the OS kernel would change, and the new kernel would fail the integrity test.

2.2 AEGIS Boot Process

Every computer with the IBM PC architecture follows approximately the same boot process. We have divided this process into four levels of abstraction (see Figure 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.

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.

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 signature. Once a valid ROM signature is found by the BIOS, control is immediately passed to it. When the ROM completes its execution, control is returned to the BIOS.

[1] Ideally, the signature would be embedded in the firmware of the ROM.

The final step of the POST process calls the BIOS operating system bootstrap interrupt (Int 19h). 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 sector into memory and passes control to it. The code contained in the boot block proceeds to load the operating system, or a secondary boot sector depending on the operating system [Gri93] [Eli96] or boot loader [Alm96].

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 1.

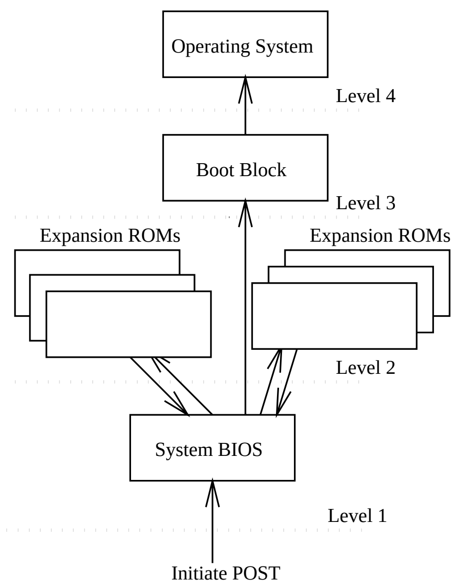


Figure 1: IBM PC boot process

2.2.1 A Layered Boot Process

We have divided the boot process into several levels to simplify and organize the AEGIS BIOS modifications, as shown in Figure 3. Each increasing level adds functionality to the system, providing correspondingly higher levels of abstraction. The lowest level is Level 0. Level 0 contains the small section of *trusted* software, digital signatures, public key certificates, and recovery code. The integrity of this level is assumed to be valid. We do, however, perform an initial checksum test to identify PROM failures. The first level contains the remainder of the usual BIOS code, and the CMOS. The second level contains all of the expansion cards and their associated ROMs, if any. The third level contains the operating system boot sector(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 public key cryptography and cryptographic hashes to protect the transition from each lower level to the next higher one, and its recovery process ensures the integrity of the next level in the event of failures. The pseudo code for the action taken at each level, L, before transition to level $L+1$ is shown in Figure 2. The function *IntegrityValid* first finds the com-

```
int IntegrityValid(Level L) {
   Certificate c = LookupCert(L);
   int result;

   if (result = VerifyCertChain(c))
      return DSAVerify(SHA1(L), c);
   else return result;
}

if (IntegrityValid(L+1))) {
   GOTO(L+1);
} else {
   Recovery(L+1);
}
```

Figure 2: **Layer Transition Pseudo code**

ponent certificate for Level L. Ideally this will be stored in the component itself, but initially it will be stored in a table contained in Level 0. Once the certificate, c, is found. *VerifyCertChain* then verifies that the certificate(s) form a "chain" of trust from the component certificate to the root Certificate Authority Public Key. If they do not, then both *VerifyCertChain* and *IntegrityValid* return an error code and a recovery procedure is entered. If *VerifyCertChain* returns TRUE, then the signature contained in the certificate is verified using the public key contained in the certificate.

Any integrity or certificate failures identified in the above process are recovered through the trusted repository.

2.3 Integrity Policy

Formalizing the discussion in Section 1.1, the AEGIS integrity policy prevents the execution of a component if its integrity can not be validated. There are three reasons why the integrity of a component could become invalid. The first is the integrity of the component could change because of some hardware or software malfunction, or it could change because of some malicious act. Finally, the component's certificate timestamp may no longer be valid. In each case, the client *MUST* attempt to recover from a trusted repository. Should a trusted repository be unavailable after several attempts, then the client's further action depends on the integrity policy of the user. For instance, a user may choose to continue operation in a limited manner, or they may choose to halt operations altogether.

2.4 Trusted Repository

The trusted repository can either be an expansion ROM board that contains verified copies of the required software, or it can be a network host. If the repository is a ROM board, then simple memory copies can repair or shadow failures. If the repository is a network host, then a protocol with strong authentication is required.

In the case of a network host, the detection of an integrity failure causes the system to boot into a recovery kernel contained on the network card ROM. The recovery kernel contacts a "trusted" repository through the secure protocol described in this paper to recover a signed copy of the failed component or its certificate. The failed component is then shadowed or repaired, and the system is restarted (warm boot).

The resultant AEGIS boot process is shown in Figure 3. Note that when the boot process enters the recovery procedure it becomes isomorphic to a secure network boot with the purpose of retrieving valid bootstrap components rather than an operating system. We leverage this fact by adding authentication to the well known network protocols supporting Remote Program Loading (RPL) DHCP[Dro97b], and TFTP[Fin84] and using them as our recovery protocol. As a result, our approach is similar to that proposed in the NetPC specification[CCP+97]. The biggest difference, however, between our approach and the NetPC approach, in addition to that noted above, is the addition of security. Currently, the NetPC specification does not contain *any* form of security. The authors of the NetPC specification, however, are developing a security architecture, and it will likely be announced by the conference date.

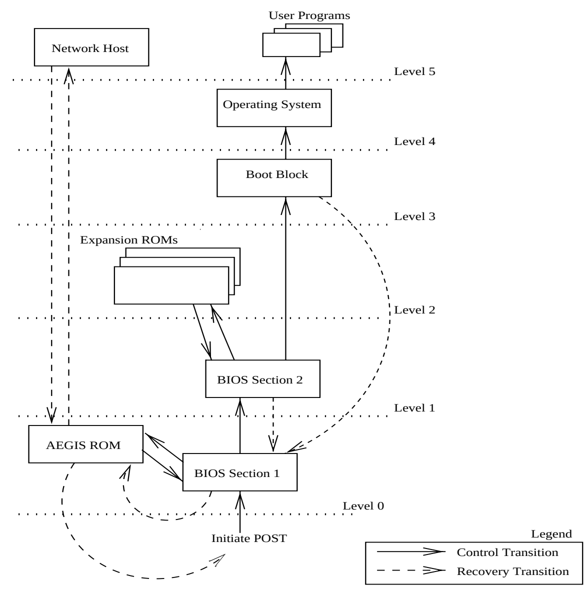


Figure 3: AEGIS boot control flow

3 AEGIS Network Recovery Protocol

The AEGIS network recovery protocol combines protocols and algorithms from networking and cryptography to ensure the security of the protocol. This section first provides an introduction to the material needed to fully understand the recovery protocol. We then describe the protocol and provide examples of its use.

3.1 Certificates

The usual purpose of a certificate with respect to public key cryptography is to bind a public key with an identity. While this binding is essential for strong authentication, it severely limits the potential of certificates, e.g. anonymous transactions. The most widely used certificate standard, the X.509[Com89] and its variants, provide *only* this binding. The X.509 standard, also, suffers from other serious problems. The most significant is ambiguity in the parsing of compliant certificates because of its use of the Basic Encoding Rules (BER)[Com88]. The encoding rules also require a great deal of space to implement, and the encoded certificates are usually large. While the V3 specification eliminates most of the problems above, the remaining ones prevent its use.

Because of the limits and problems with the X.509 certificate standard, we use a subset of the proposed SDSI/SPKI 2.0 certificate structure[EFRT97][Ell97] instead. The SDSI/SPKI format does not suffer from the same problems as X.509, and it offers additional functionality.

3.1.1 SDSI/SPKI Lite

Since the SDSI/SPKI standard is still under development, we have chosen to support the small subset of SDSI/SPKI needed for AEGIS. We call this subset SDSI/SPKI Lite.

SDSI/SPKI provides for functionality beyond the simple binding of an identity with a public key. Identity based certificates require the existence of an Access Control List (ACL) which describe the access rights of an entity. Maintaining such lists in a distributed environment is a complex and difficult task. In contrast, SDSI/SPKI provides for the notion of a capability [Lev84]. In a capability based model, the certificate carries the authorizations of the holder eliminating the need for an identity infrastructure and access control lists. In AEGIS, we use two capabilities: SERVER, and CLIENT with the obvious meanings.

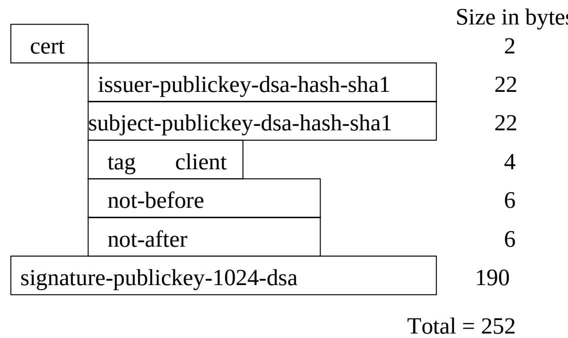
In AEGIS we only use three types of certificates. The first is an authorization certificate. This certificate, signed by a trusted third party or certificate authority, grants to the key holder (the machine that holds the private key) the capability to generate the second type of certificate- an authentication certificate. The authentication certificate demonstrates that the client or server actually hold the private key corresponding to the public key identified in the authentication certificate. A nonce field is used along with a corresponding nonce in the server authentication certificate to ensure that the authentication protocol is "Fail Stop"[GS95] detecting and preventing active attacks such as a man–in–the–middle. The *msg-hash* field ensures that the entire message containing the certificates has not been modified. Using the *msg-hash* in the authentication certificate eliminates a signature and verification operation since the entire message no longer needs to be signed. The additional server fields are used to pass optional Diffie-Helman parameters to the client so that these parameters need not be global values. While clients are free to set the validity period of the authentication certificate to whatever they desire, we expect that clients will keep the period short.

The current SDSI/SPKI draft RFC proposes several encoding schemes. The one shown in Figures 5, 6, and 7 is the Advanced Transport Format (ATF). Basically, an ASCII representation of the certificate with binary information, e.g. keys, represented by Base64 encoding. Unfortunately, none of the proposed representation schemes for SDSI/SPKI produce certificates small enough for our purposes. Therefore, we propose a new encoding scheme which we call the Binary Transport Format (BTF). BTF uses fixed identifiers for the various certificate types, and a direct mapping of binary data into its network byte order representation. The resulting encoding scheme reduces the size of a certificate from approximately one thousand bytes when encoded in ATF to slightly less than two hundred and fifty six bytes when encoded in BTF. Space is also saved by combining several SPKI fields into a single type, e.g. *(subject (hash-of-key (hash sha1 bytes)))* maps to *subject-publickey-dsa-hash-sha1*. While this expands the name space significantly, it also saves a significant amount of space. Table 1 lists sample type identifiers for BTF. BTF uses two formats. The first is a three tuple of identifier, size, data. The identifier is two bytes, the size is two bytes, and the data is a variable size. The second is an implied size format based on the identifier. The latter format is used as much as possible to save space.

Examples of these certificates are shown in Figures 4

```
((cert (issuer (hash-of-key (hash sha1 cakey)))
        (subject (hash-of-key (hash sha1 keyholderkey)))
        (tag (client))
        (not-before 03/29/97-0000)
        (not-after 03/29/98-0000)
   (signature (hash sha1 hashbytes)
      (public-key dsa-sha1 cakey)
      (sigbytes)))
```

(a) ATF



(b) BTF

Figure 4: AEGIS Authorization Certificate

SPKI Type	Identifier
cert	0xaeba
issuer	0x0001
issuer-publickey-dsa-hash-sha1	0x1001
subject	0x0002
subject-publickey-dsa-hash-sha1	0x1002
sha1-hash	0x0004
tag	0x0005
not-before	0x0006
not-after	0x0007
signature	0x0008
signature-publickey-1024-dsa-hash-sha1	0x1008
signature-publickey-1024-dsa	0x2008

Table 1: Sample Binary Transport Format Identifiers

```
((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 5: AEGIS Client Authentication Certificate

, 5, and 6. The first figure shows an authorization certificate in both ATF and BTF forms. The remaining Figures use only ATF for readability purposes. The third and final certificate format is the component signature certificate shown in Figure 7. This certificate is either embedded in a component or stored in a table. It is used with the AEGIS boot process described earlier in this paper.

3.1.2 Certificate Revocation Lists

Requiring each client to maintain a Certificate Revocation List (CRL) places a significant burden on the non-volatile

```
((cert (issuer (hash-of-key (hash sha1
                serverkey)))
    (subject (hash-of-key (hash sha1
                  serverkey)))
     (tag (server (dh-g gbytes)
             (dh-p pbytes)
             (dh-Y ybytes)
             (msg-hash
              (hash sha1 hbytes))
             (cnonce cbytes)
             (snonce sbytes)))
     (not-before 09/01/97-0900)
     (not-after 09/01/97-0900))
 (signature
   (hash sha1 hashbytes)
   (public-key dsa-sha1 serverkey)
   (sigbytes)))
```

Figure 6: AEGIS Server Authentication Certificate

```
((cert (issuer (hash-of-key (hash sha1
                approverkey)))
     (subject (hash sha1
                 hashbytes))
     (not-before 09/01/97-0000)
     (not-after 09/05/97-0000))
 (signature (hash sha1
                 hashbytes)
     (public-key dsa-sha1
                 approverkey)
     (sigbytes)))
```

Figure 7: AEGIS Component Certificate

storage of the client. Rather than use CRLs, we choose instead to keep the validity period of certificates short as in the SDSI/SPKI model and require the client to update the certificates when they expire. This serves two purposes beyond the ability to handle key revocation. First, we eliminate the storage requirements for CRLs which would overburden a client. Second, we can potentially reduce the amount of system maintenance required of the client. Since the client must connect to the server on a regular basis to update the component certificates, the server can, at the same time, update the actual component as well if a new version is available.

3.2 Diffie Hellman Key Agreement

The Diffie Hellman Key Agreement (DH) [DH76] permits two parties to establish a shared secret between them. 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 as proposed in the Station to Station Protocol (StS)[DvOW92]. Our recovery protocol is an extension to StS.

The DH 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 $mod\ n$. 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 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.

3.3 Digital Signature Standard

The Digital Signature Standard (DSS) includes a digital signature algorithm (DSA) [oS94] and a cryptographic hash algorithm (SHA1) [oS95]. DSA produces a 320 bit signature using the following parameters:

A prime, p, between 512 and 1024 bits in length. The size of the prime must also be a multiple of 64.

A 160 bit prime factor, q, of $p - 1$.

g, where $g = h^{(p-1)/q} \mod p$ and h is less than $p - 1$ such that g is greater than 1.

x, where x is less than q.

y, where $y = g^x \mod p$.

The parameters p, q, and g are public. The private key is x, and the public key is y.

A signature of a message, M, is computed in the following manner. The signer generates a random number, k, that is less than q, and then computes $r = (g^k \mod p) \mod q$ and $s = (k^{-1}(SHA1(M) + xr)) \mod q$. The values r and s, each 160 bits in length, comprise the signature. The receiver verifies the signature by computing:

$w = s^{-1} \mod q$

$u_1 = (SHA1(M) * w) \mod q$

$u_2 = (r * w) \mod q$

$v = ((g^{u_1} * y^{u_2}) \mod p) \mod q$.

The signature is verified by comparing v and r. If they are equal, then the signature is valid.

3.4 IPSEC Authentication Header

The IPSEC Authentication Header (AH) provides authentication and integrity of an IP datagram [KA97]. The format for the IPSEC Authentication Header is shown in Figure 8. The *next header* field describes the type of header following AH. The *Length* field indicates the size of the header in 4-byte units minus 2. For instance, if the entire header size was 192 bits then the length field would have a value of 4. The Security Parameters Index (SPI) determines which security association defined between the source and destination to use. The sequence number is a 32 bit field that is used to prevent replay attacks, and the *authentication data* field is a variable length, aligned to 32 bits, field containing the appropriate authentication information. In the case of AEGIS, this is a 96 bit MAC.

0	8	16	31
Next Hdr	Length	Reserved	
Security Parameters Index			
Sequence Number Field			
Authentication Data (variable)			

Figure 8: IPSEC AH Format

3.5 SHA1 Message Authentication Code

Message Authentication Codes (MAC) utilize a secret, k, shared between the communicating parties and a message digest. We use the Secure Hash Algorithm (SHA1), and the HMAC described in RFC 2104[KBC97] and a draft RFC[MG97]. The MAC is defined as:

$$SHA1(k \text{ XOR } opad, SHA1(k \text{ XOR } ipad, M)),$$

where M is the message or datagram, *opad* is an array of 64 bytes each with the value 0x5c, and $ipad$ is an array of sixty four bytes each with the value 0x36. k is zero padded to sixty four bytes. The result of this MAC is the 160-bit SHA1 digest which is truncated to the first ninety six bits. These bits are used as the MAC.

3.6 Dynamic Host Configuration Protocol

The DHCP protocol[Dro97b] provides clients the ability to configure their networking and host specific parameters dynamically during the boot process. 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 which is described in Section 4.1.

The format of a DHCP message is shown in Figure 9[Dro97b]. The first field in the DHCP 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*

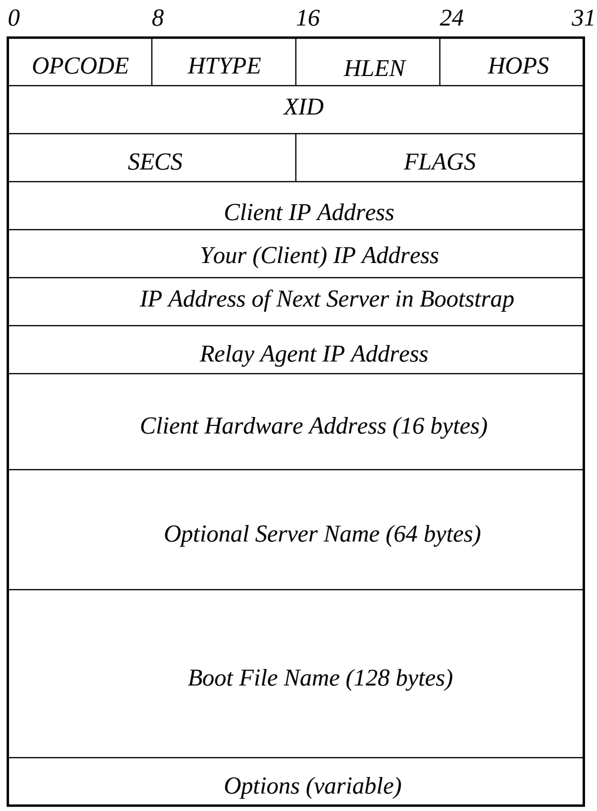


Figure 9: DHCP Message Format

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. Its use is 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 (or had a bad one) its address. *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 10.

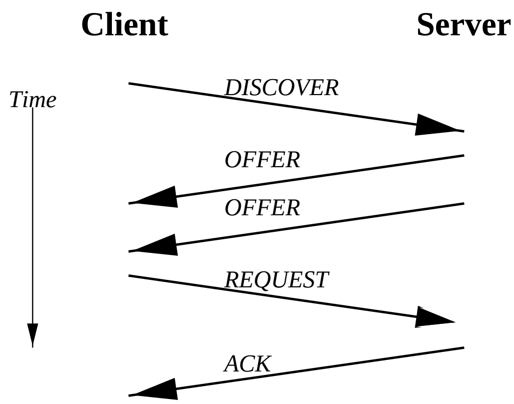


Figure 10: Initial DHCP Message Exchange

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.

3.7 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.

3.8 Initial Mutual Authentication Protocol

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_{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 CA on the authorization certificate is also verified (or chain of certificates). 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, and Y, where $Y = g^y \mod p$. If the DH parameters are not identified in the certificate, then default values for g and p are used. Currently, we are using 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 that 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 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

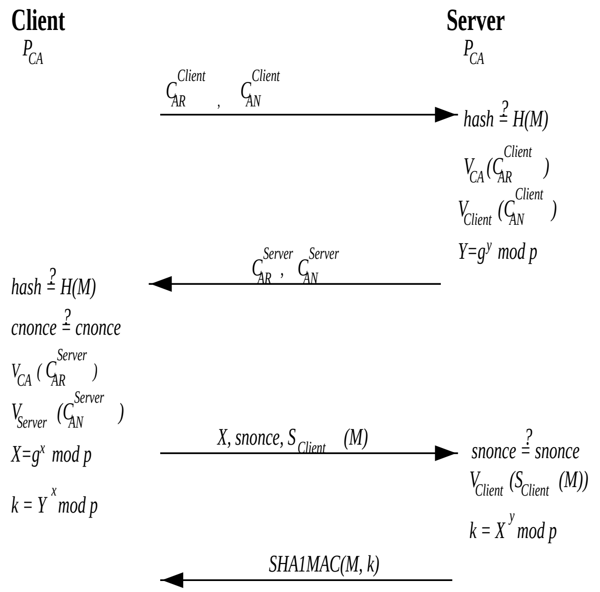


Figure 11: Authentication Message Exchange

such time as both agree to change k. Figure 11 depicts the entire exchange between the Client and the Server with the DHCP messages identified. The use of 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. This reduces 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 a benefit not possible with basic X.509 certificates.

3.9 Subsequent Message Authentication

After the establishment of the shared secret through the protocol described above, subsequent DHCP messages are authenticated through the use of the SHA1 HMAC defined in Section 3.5 augmented with a one up counter to prevent replays. The counter is initially set to zero when the shared secret, k, is derived. In computing the MAC, the fields *giaddr* and *hops* must be zeroed since these fields are mutable by relay agents.

Authentication of the subsequent TFTP messages require the use of the IPSEC Authentication header option described in Section 3.4.

4 Implementation

Moving from a high level design to an implementation requires a great deal of work. In this section we take the protocol and certificates described in section 4 and describe their implementation using DHCP and TFTP. We also provide the message formats and type information. We conclude the section by providing performance information, and discussing related work.

4.1 DHCP Authentication Option

DHCP is extensible through the use of the variable length options field at the end of each DHCP message. The format and use of this field is currently defined by an Internet RFC [AD97]. An option for authentication is also defined by a draft RFC [Dro97a]. The format of the authentication option is shown in Figure 12. The DHCP authentication

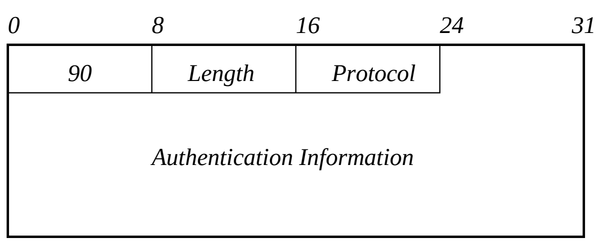


Figure 12: DHCP Authentication Option Format

option was designed to support a wide variety of authentication schemes by using single byte protocol and length fields. Unfortunately, a single byte value for the size in octets of authentication information severely limits the ability to include public key certificates, with reasonable key sizes, in the data field of the option. Fortunately however, we do so by using the BTF format described earlier, and using multiple authentication option fields. While this latter approach *technically* violates the DHCP authentication option protocol, it does not cause any interoperability

Type	Value
Authorization Certificate	0
Client Authentication Certificate	1
Server Authentication Certificate	2
Component Authentication Certificate	3
X value	4
snonce	5
signature	6
SHA1MAC	7

Table 2: AEGIS Types

problems. An alternate approach would have required increasing the the option size field from one to two bytes. While interoperability issues could be mitigated, the approach still presented a significant change to the DHCP protocol.

Since we must use multiple authentication option fields in a DHCP message, we must add a field to identify the information contained in the option. The resultant AEGIS authentication option format is shown in Figure 13, and a table describing the various types is shown in Table 2. The client and server use this option format to exchange

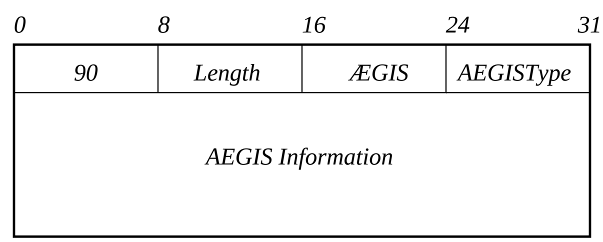


Figure 13: AEGIS Authentication Option Format

the information required by our protocol.

In addition to using BTF formatted SPKI certificates, we support the use of a new DHCP option to permit the continuation of the previous option field. Through the use of this option, any information that exceeds the two hundred and fifty six bytes available in a DHCP option can be extended into the next field. This permits the use of X.509v3 certificates if desired.

IP HDR	AH HDR	UDP HDR	TFTP FRAME

Figure 14: Authenticated TFTP Datagram Format

4.2 Trivial File Transfer Protocol Authentication

As we discussed earlier, adding authentication to TFTP packets within the currently defined protocol is problematic. Rather than suffer the potential interoperability problems, we use the IPSEC Authentication header[KA97]. This approach has several benefits. The foremost is that it doesn't require any changes to the TFTP protocol itself, and it uses a proposed standard for authentication. The resulting TFTP datagram is shown in Figure 14.

4.3 Using DHCP/TFTP as the Recovery Protocol

Once authentication is added to DHCP and TFTP, AEGIS can use them without further modifications as its recovery protocol. In AEGIS, the client follows the DHCP protocol but adds to the DHCPDISCOVER message the name of the required component needed followed by the SHA1 hash of the component in the boot file name field. Once the DHCP protocol is completed and the shared secret established, the AEGIS client contacts the trusted repository using TFTP with authentication and downloads the new component.

4.4 Prototype Information

We are currently in the process of completing this work using the Internet Software Consortium's DHCP server [Lem97], the GNU Multi-Precision Arithmetic package (GnuMP) [Gra96], the Etherboot package [GY97], and portions of AT&T's Cryptolib [LMB95]. Currently, the prototype ROM image is approximately 41 KB. This includes a subset of the DHCP protocol and all of the cryptographic code described in this paper. It currently does not contain the BTF certificate parsing code.

Algorithm	Time
SHA1	6.1 MB/sec
DSA Verify (1024bit)	88 msec
DSA Sign (1024bit)	26 msec
Generate X,Y (1024bit)	27 msec
Generate k (1024bit)	91 msec

Table 3: AEGISrom Cryptographic Benchmarks

4.4.1 Performance Information

This section provides performance estimates for the recovery protocol based on its current implementation. The timings were obtained using a 200Mhz Pentium with 32MB of main memory using an Intel EtherExpress Pro100B network interface card and the AEGIS recovery ROM code. The SHA1 code is from Cryptolib 2.0beta, and the DSA and Diffie-Hellman implementations were done using GNU MP. Times for the cryptographic operations are shown in Table 3. The times for DSA Verify and Sign include the cost of computing the SHA1 digest of the resultant DHCP message.

4.4.2 Initial Exchange

The initial authentication exchange includes the first three DHCP messages, *DHCPDISCOVER*, *DHCPOFFER* and *DHCPREQUEST*. *DHCPDISCOVER* requires the client to perform one signature operation, and the server must perform two verify operations. Thus, the total cost of this message is 202 msec. The *DHCPOFFER* message requires the server to generate Y and perform one signature operation. The client must perform two verify operations. This results in a message cost of 229 msec. The final message, *DHCPREQUEST*, requires the client to generate X and k, and perform one signature operation. The server must perform one verify operation, and generate k resulting in a message cost of 323 msec. Summing the cost of these three messages gives a total cost of 754 msec.

While the above time may seem too high a cost to pay for security, the total time is small when compared to the total time spent booting a computer system. It is unlikely that users will see the increase in time required to perform the authentication. Also, the above times are unoptimized at this point in the prototype.

4.4.3 Subsequent Exchanges

Subsequent DHCP and TFTP messages use the MAC described earlier, and will likely (in a LAN situation) be bounded by the speed of SHA1, 6.1 MB/sec.

4.5 Related Work

To our knowledge, there is no previous academic work involving the secure recovery of bootstrap components. Recently, however, several commercial products have been announced that allow system administrators to automatically update and manually repair bootstrap components. None of the proposed products, however, include automatic recovery and repair.

There are several efforts at incorporating authentication into DHCP. Microsoft, Intel and others are working on developing a security architecture for the NetPC specification which uses DHCP and TFTP. While an early draft of this paper was provided to members of that group, the group has not revealed their draft architecture yet.

There are also two draft RFCs. The first effort [Dro97a] involves the use of a shared secret between the DHCP client and server. While this approach is secure, it severely limits the mobility of clients to only those domains where a shared secret was previously established. Furthermore, the maintenance and protection of the shared secrets is a difficult process. Another effort at incorporating authentication into DHCP is by TIS. This proposal combines DHCP with DNSSEC[EK97]. This approach provides for the mobility of DHCP clients, but at a significant increase in cost in terms of complexity. The client implementation, in order to support this approach, must also include an implementation of DNSSEC. This will significantly increase the size of client code- possibly beyond the ROM size available to the client. Recently, Intel has proposed authentication support for DHCP [Pat97]. Their proposal uses a two phase approach. In the first phase, the computer system boots normally using DHCP. The second phase begins after the system completes the DHCP process and uses ISAKMP [MSST96] to exchange a security association. This security association is then used to once again obtain the configuration information from the DHCP server using a secure channel, if such a channel can be established. This information is then compared to that obtained in the first phase. If they differ or a secure channel cannot be established, then the boot fails. The benefit of this approach is that it requires no changes to DHCP. The drawbacks are the same as the DNSSEC approach with the addition of two problems. The first is a possible race condition vulnerability during the time before the two configurations are compared. The second is that the approach does not protect against denial of service attacks.

5 Future Work

One of the major goals of the AEGIS research has been the development of new ideas for the construction of secure systems, with the additional constraint that the ideas must be realizable today or in the very near term with commercial platforms. While confining, this constraint ensures that AEGIS results will have impact beyond simply the academic community.

We intend to further investigate the centralized management of the bootstrap process. This has many practical uses, including desktop management in LAN-attached PCs (where integrity failures might be stimulated by viruses or user-inserted cards), as well as secure, recoverable bootstrap for network elements with processors, such as bridges, IP routers, and “Active Networks”[AAKS97].

The recovery protocol itself will be fully incorporated into the DHCP model, and we intend to propose it as an authentication RFC standard. We also intend to propose the DHCP option continuation as a standard. We expect to make these proposals at the December 1997 Internet Engineering Task Force meeting.

6 Conclusions

We introduced the AEGIS secure bootstrap architecture, explained its approach to integrity and the assumptions it makes about the operating environment, and discussed the general idea behind automated recovery in a secure bootstrap process using a trusted repository. We are currently implementing this new automated recovery process in the context of the PC architecture using a small portion of the BIOS. We have shown how it can be extended to recovery over networks by use of cryptographic protocols, and provided one such protocol, with expected data structures and packet formats.

We believe that this work has a significant impact on the administration and manage-ability of systems. While we have previously demonstrated the need and provided an architecture for a secure bootstrap for any trusted system, here we have shown how that architecture can be utilized in a very realistic environment, with no loss of security. Thus, we 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.

References

[AAKS97] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A Secure Active Network Environment Architecture. Technical Report MS-CIS-97-17, University of Pennsylvania, November 1997.

[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.

[Alm96] Werner Almesberger. *LILO Technical Overview*, version 19 edition, May 1996.

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

[CCP+97] Compaq Computer, Dell Computer, Hewlett Packard, Intel, and Microsoft Corporations. *Network PC System Design Guidelines*, 1.0b edition, July 1997.

[Com88] Consultation Committee. *Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, 1988.

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

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

[Dro97a] R. Droms. Authentication for DHCP Messages. Work in Progress, August 1997.

[Dro97b] R. Droms. Dynamic Host Configuration Protocol, 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.

[EK97] D. Eastlake and C. Kaufman. Dynamic Name Service and Security. Internet RFC 2065, January 1997.

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

[Ell97] Carl M. Ellison. SDSI/SPKI BNF. Private Email, July 1997.

[Fin84] Ross Finlayson. Bootstrap Loading using TFTP. Internet RFC 906, June 1984.

[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.

[Gol97] Joe Goldmeer. Re: Re: Pentium pro floating point patch. USENET Posting to intel.motherboards.pentium_pro, November 1997.

[Gra96] Torbjorn Granlund. *The GNU Multiple Precision Arithmetic Library*. TMG Datakonsult, 2.0.2 edition, June 1996.

[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.

[GY97] Markus Gutschke and Ken Yap. Etherboot 3.2. http://www.syd.dit.csiro.au/staff/ken/personal/etherboot/, July 1997.

[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.

[KA97] Stephen Kent and Randall Atkinson. Work in Progress, October 1997.

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

[Lem97] Ted Lemon. Dynamic Host Configuration Server. ftp://ftp.fugue.com/pub/, 1997.

[Lev84] H.M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[LMB95] Jack Lacy, Don Mitchell, and Matt Blaze. Cryptolib 2.0beta. Email to cryptolib@research.att.com, 1995.

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

[MG97] C. Madson and R. Glenn. The Use of HMAC-SHA1-1-96 within ESP and AH. Work in Progress, November 1997.

[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.

[MSST96] Douglas Maughan, Mark Schertler, Mark Schneider, and Jeff Turner. Internet Security Association and Key Management Protocol (ISAKMP). Internet–draft, IPSEC Working Group, June 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).

[Pat97] Baiju V. Patel. Securing DHCP. Work in Progress, July 1997.

[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.

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

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