A Secure and Reliable Bootstrap Architecture[*†]

William A. Arbaugh[‡]
David J. Farber
Jonathan M. Smith
University of Pennsylvania

December 2, 1996

Abstract

In a computer system, the integrity of lower layers is treated as axiomatic by higher layers. Under the presumption that the hardware comprising the machine (the lowest layer) is valid, 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.

When these conditions are not met, as they typically are not in the bootstrapping (initialization) of a computer system, no integrity guarantees can be made. Yet, these guarantees are increasingly important to diverse applications such as Internet commerce, intrusion detection systems, and "active networks." In this paper, we describe the AEGIS architecture for initializing a computer system. It validates integrity at each layer transition in the bootstrap process. AEGIS also includes a *recovery* process for integrity check failures, and we show how this results in robust systems. We discuss our prototype implementation for the IBM personal computer (PC) architecture, and show that the cost of such system protection is surprisingly small.

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 levels of abstraction, 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*. Without integrity, no system can be made secure.

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 [20]. An essential presumption of the security arguments for these designs was that system layers underpinning the operating system, whether hardware, firmware, or both, are trusted. We find it surprising, given the great attention paid to operating system security [13] [8] that so little attention has been paid to the underpinnings required for secure operation, *e.g.,* a secure bootstrapping phase for these operating systems.

Without such a secure bootstrap the operating system kernel cannot be trusted since it is invoked by an untrusted process. Designers of trusted systems often avoid this problem by including the boot components in the trusted computing base (TCB) [6]. That is, the bootstrap steps are explicitly trusted. We believe that this provides a false sense of security to the users of the operating system, and more important, is unnecessary.

1.1 AEGIS

We have designed 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

[†]Submitted to the 1997 IEEE Security and Privacy Conference.
[‡]Also with the U.S. Department of Defense

digital signature associated with each component.

The AEGIS architecture includes a recovery mechanism for repairing integrity failures which protects against some classes of denial of service attacks. 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.

1.2 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 source* 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.

1.3 Outline of the paper

In Section 2, we make the assumptions of the AEGIS design explicit. Section 3 is the core of the paper, giving an overview of the AEGIS design, and then plunging into details of the IBM PC boot process and its modifications to support AEGIS. A model and logical dependencies for integrity chaining are given in Section 4, and a calculation of the complete bootstrap performance is given; performance is surprisingly good. Section 5 discusses related work and critically examines some alternative approaches to those taken in AEGIS. We discuss the system status and our next steps in Section 6, and conclude the paper with Section 7.

2 Assumptions

The first assumption upon which the AEGIS model is based is that 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. We also depend on the integrity of an expansion card which contains copies of the essential components of the boot process for recovery purposes, cryptographic certificates, and optionally a small operating system for recovering components from a trusted network host.

The second assumption is the existence of a cryptographic certificate authority infrastructure in order to bind an identity with a public key. However, there is no restriction on its form, *e.g.,* single trusted authority, hierarchical, web of trust [22] [3].

The final assumption is that some trusted source exists for recovery purposes. This source may be a host on a network that is reachable through a secure communications protocol, or it may be the trusted ROM card located on the protected host.

3 AEGIS Architecture

3.1 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 block 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 do not substantially increase the size of the boot block.

AEGIS modifies the boot process shown in figure 2 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 the addition of an inexpensive PROM board, and modifications to the BIOS. The BIOS and the PROM board contain the verification code, and public key certificates. The PROM board also contains code that allows the secure recovery of any integrity failures found during the initial bootstrap. In essence, the trusted software serves as the root of an authentication chain that extends to the operating system and potentially beyond to applica-

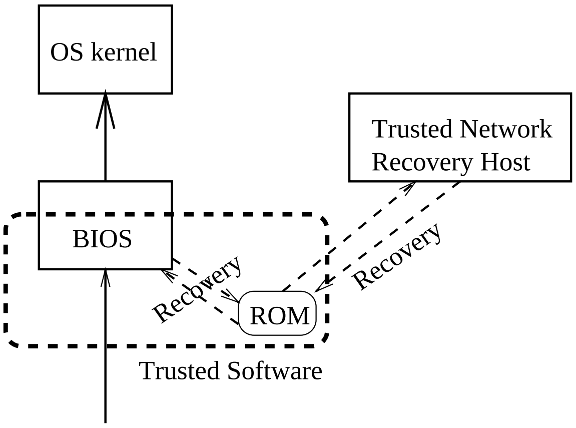

Figure 1: AEGIS boot overview

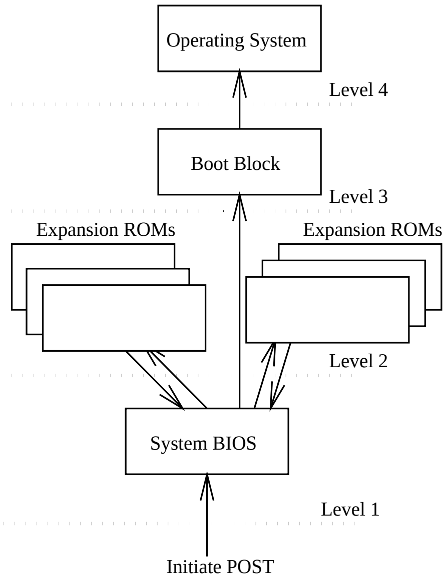

Figure 2: IBM PC boot process

tion software [18] [9] [15]. A high level depiction of the bootstrap process is shown in figure 1. In the AEGIS boot process, either the operating system kernel is started, or a recovery process is entered in order to repair any integrity failure detected. Once the repair is completed, the system is restarted to ensure that the system boots. This entire process occurs without user intervention.

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 signature for each expansion card, any additional expansion cards will fail the integrity test. Similarly, a new operating system cannot be started since the boot block would change, and the new boot block would fail the integrity test.

3.2 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 2), which correspond to phases of the bootstrap operation. The first phase is the Power on Self Test or POST [17]. 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.

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 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 [10] [7].

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.

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.

3.2.1 A Multilevel 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 in order to identify PROM failures. The first level contains the remainder of the usual BIOS code. The second level contains all of the expansion 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 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.

3.2.2 AEGIS BIOS Modifications

AEGIS modifies the boot process shown in figure 2 by dividing the BIOS into two logical sections. The first section contains the bare essentials needed for integrity verification and recovery. Coupled with the AEGIS ROM, it comprises the “trusted software”. The second section contains the remainder of the BIOS.

The first section executes and performs the standard checksum calculation over its address space in order to protect against ROM failures. Following successful completion of the checksum, the cryptographic hash of the second section is computed and verified against a stored signature. If the signature is valid, control is passed to the second section, *i.e.,* Level 1.

The second section proceeds normally with one change. Prior to executing an expansion ROM, a cryptographic hash is computed and verified against a stored digital signature for the expansion code. If the signature is valid, then control is passed to the expansion ROM. Once the verification of each expansion ROM is complete (Level 2), the BIOS passes control to the operating system bootstrap code. The bootstrap code was previously verified as part of the BIOS, and thus no further verification is required. The bootstrap code finds the bootable device and verifies the boot block.

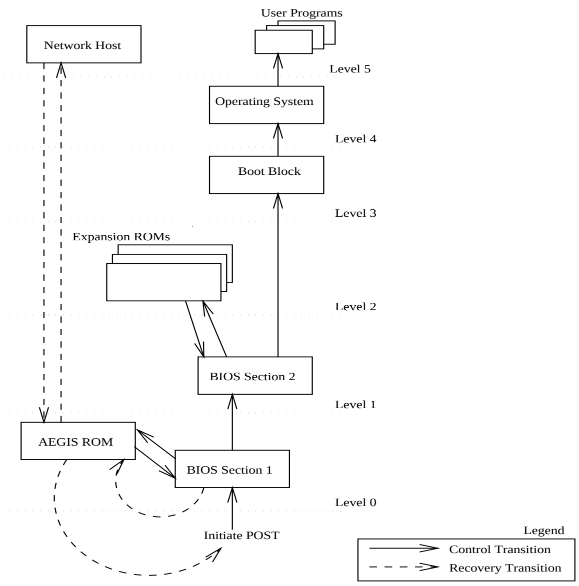

Figure 3: AEGIS boot control flow

Assuming that the boot block is verified successfully, control is passed to it (Level 3). If a secondary boot block is required, then it is verified by the primary block before passing control to it. Finally, the kernel is verified by the last boot block in the chain before passing control to it (Level 4).

Any integrity failures identified in the above process are recovered either through storage on the expansion ROM card, or through a network host. If the component that fails its integrity check is a portion of the BIOS, then it must be recovered from the ROM card. The recovery process is a simple memory copy from the address space of the ROM card to the memory address of the failed component, in effect shadowing the failed component.

A failure beyond the BIOS causes the system to boot into a recovery kernel contained on the ROM card. The recovery kernel contacts a “trusted” host through a secure protocol, *e.g.,* IPv6 [1], to recover a verified copy of the failed component. The failed component is then replaced and the system is restarted.

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.

4 Integrity Chaining and System Performance

In AEGIS, system integrity is preserved through the chain of integrity checks in the boostrap process. The ideal authentication chain produced by each level verifying the next can be represented by the recurrence

$$I_0 = True, \qquad I_{i+1} = \left\{ I_i \bigwedge V_i(L_{i+1}) \quad \text{for } 0 < i \leq 4. \right. \tag{1}$$

I_i is a boolean value representing the integrity of level i, and $\bigwedge$ is the boolean *and* operation. V_i is the verification function associated with the i^{th} level. V_i takes as its only argument the level to verify, and it returns a boolean value as a result. The verification function performs a cryptographic hash of the level, and compares the result to the value obtained from a stored signature for the level. As stated earlier, the IBM PC does not lend itself to such a boot process. Instead, we alter the recurrence to:

$$I_0 = \text{True}, \qquad I_{i+1} = \begin{cases} I_i \bigwedge V_i(L_{i+1}) & \text{for } i = 0, 3, 4, \\ I_i \bigwedge \sum_{l=1}^{n} V_i(L_{i+1}^l) & \text{for } i = 1, \\ I_i \bigwedge V_{i-1}(L_{i+1})) & \text{for } i = 2. \end{cases} \tag{2}$$

Here, n represents the number of expansion boards in the system, and our level of assurance is preserved.

4.1 Performance impact on bootstrap completion time

Using the recurrence relation shown in equation 2, we can compute the estimated increase in boot time ($T_\triangle$), without integrity failures, between AEGIS and a standard IBM PC using the following equation:

$$T_\triangle = t(V_0(L_1)) + t(\sum_{l=1}^{n} V_1(L_2^l)) + t(V_1(L_3)) + t(V_3(L_4)), \tag{3}$$

where $t(op)$ returns the execution time of op. In estimating the time of the verification function, V_i, we use the BSAFE benchmarks [19] for an Intel 90Mhz Pentium computer, shown in table 1. The cost of verification includes time required for computing a MD5 message digest, and the time required to verify the digest against a stored signature. Any signatures embedded in the public key certificate are ignored at the moment.

Algorithm	Time
MD5	13,156,000 bytes/sec
RSA Verify (512bit)	0.0027 sec
RSA Verify (1024bit)	0.0086 sec
RSA Verify (2048bit)	0.031 sec

Table 1: BSAFE 3.0 Benchmarks

The BIOS is typically one megabit (128 Kilobytes), and the expansion ROMs are usually 16 kilobytes with some, such as video cards, as large as 64 kilobytes. For analysis purposes, we will assume that one 64 kilobyte card and two 16 kilobyte cards are present. The size of the boot blocks for FreeBSD 2.2 (August 1996 Snapshot) are 512 bytes for the primary boot block, 6912 bytes for the secondary boot block, and 1,352 kilobytes for the size of the GENERIC kernel. Using the performance of MD5 from table 1, the time required to verify each layer using a 1024 bit modulus are:

$$\begin{aligned} t(V_0(L_1)) &= 0.0185 seconds \\ t(V_1(L_2)) &= 0.0160 seconds \\ t(V_1(L_3)) &= 0.018 second \\ t(V_3(L_4)) &= 0.114 seconds. \end{aligned}$$

Summing these times gives $T_\triangle = 0.1665 seconds$ which is insignificant compared to the length of time currently needed to bootstrap an IBM PC.

5 Related work

The first presentation of a secure boot process was done by Yee [21]. In Yee's model, a cryptographic coprocessor is the first to gain control of the system. Unfortunately, this is not possible without a complete architectural revision of most computer systems—even if the coprocessor is tightly coupled. Yee expands his discussion of a secure boot in his thesis [23], but he continues to state that the secure coprocessor should control the boot process verifying each component prior to its use. Yee states that boot ROM modifications *may* be required, but since a prototype secure boot process was never implemented more implementations questions are raised than answered by his discussion.

Clark [5] presents a secure boot process for DOS that stores all of the operating system bootstrap code on a PCMCIA card. He does not address the verification of any firmware (system BIOS or expansion cards). Clark's model, however, does permit mutual cryptographic authentication between the user and the host which is an important capability. However, the use of a PCMCIA card containing all of the system boot files creates several configuration management problems, *e.g.,* a system upgrade requires the reprogramming of all the cards in circulation.

Lampson [12] 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. The Birlix [11] Security Architecture proposes a model designed by Michael Gross that is similar to Lampson's. The Birlix model also suffers from the same problems. In both cases, the boot ROM is responsible for generating a public and private key pair for use in host based authentication once the operating system is running. In AEGIS we leave any security related functions, beyond the boot process, to the operating system without loss of security. To do otherwise limits security choices for the operating system.

None of the approaches address a recovery process in the event of an integrity failure.

5.1 Discussion and alternative approaches

A possible criticism of this work is that booting from a floppy disk provides the same level of protection. There are several reasons why this is not so. The first is that providing physical security for the floppy disk is extremely difficult. Users can take the disks wherever they like, and do whatever they like to them. One can envision a user building their own boot floppy that gives them system level privileges. The user is now free to read and write anywhere on the local disk circumventing any security systems put in place by the "real" boot floppy or the on disk operating system. This problem is described by Microsoft [16] as a method of circumventing the Windows NT file system (NTFS). The major shortcoming, however, in using a boot disk is that none of the firmware is verified prior to use. Thus, a user can add or replace expansion boards into the system without any security controls, potentially introducing unauthorized expansion cards.

6 Status and Future Work

The AEGIS prototype is nearing completion, and we are confident that a complete description of its performance and implementation will be provided at the conference. Difficulty in obtaining BIOS source code has been a roadblock to modifying it to support AEGIS as described in the body of the paper. We have reached an agreement with a BIOS vendor to provide the source code after some legal details are finalized.

The current recovery kernel prototype uses IPv6 as a means of recovering replacement files. We intend to switch to the Internet Engineering Task Force's (IETF) Internet Security Association and Key Management Protocol (ISAKMP) [14] to allow user choice of a secure protocol. Additionally, the method with which the recovery kernel contacts a host is currently via a fixed address. We hope to develop or use a protocol in which the recovery host's address can be determined when needed.

The process by which components are vetted, signed, and the resultant signature and public key certificate installed needs to be addressed carefully. We plan to address this once a full prototype is completed, and will report on the results. As a minimum, we expect to use flaw detection techniques such as those from Bishop [2], Kannan [4], and others to assist in a technical vetting before the actual signing of the component.

We are also investigating the use of a cryptographic sideboard as a high end solution to improve performance and increase security.

7 Conclusions

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.

References

[1] Atkinson, R. J., McDonald, D. L., Phan, B. G., Metz, C. W., and Chin, K. C. Implementation of ipv6 in 4.4 bsd. In *Proceedings of the 1996 USENIX Technical Conference* (January 1996), USENIX, pp. 113–125.

[2] Bishop, M., and Dilger, M. Checking for race conditions in file accesses. *Computing Systems 9*, 2 (Spring 1996), 131–152.

[3] Blaze, M., Feigenbaum, J., and Lacy, J. Decentralized Trust Management. In *IEEE Conference on Security and Privacy* (May 1996), IEEE.

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

[5] Clark, P. C. *BITS: A Smartcard Protected Operating System.* PhD thesis, George Washington University, 1994.

[6] DOD. Trusted computer system evaluation criteria. Tech. Rep. DOD 5200.28-STD, Department of Defense, December 1985.

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

[8] Engler, D. R., Kaashoek, M. F., and Jr., J. W. O. The operating system kernel as a secure programmable machine. In *Proceedings of the Sixth SIGOPS European Workshop* (September 1994), pp. 62–67.

[9] G. Davida, Y. D., and Matt, B. Defending systems against viruses through cryptographic authentication. In *1989 IEEE Symposium on Security and Privacy* (1989), IEEE, pp. 312–318.

[10] Grimes, R. At386 protected mode bootstrap loader. /sys/i386/boot/biosboot/README.MACH, October 1993. 2.1.5 FreeBSD.

[11] Härtig, H., Kowalski, O., and Kühnhauser, W. The Birlix security architecture. *Journal of Computer Security 2*, 1 (1993), 5–21.

[12] Lampson, B., Abadi, M., and Burrows, M. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems v10* (November 1992), 265–310.

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

[14] Maughan, D., Schertler, M., Schneider, M., and Turner, J. Internet security association and key management protocol (isakmp). Internet–draft, IPSEC Working Group, June 1996.

[15] Microsoft. Authenticode techonology. Microsoft's Developer Network Library, October 1996.

[16] Microsoft. Overview of fat, hpfs, and ntfs file systems. Knowledge Base Article Q100108, Microsoft, October 1996.

[17] Phoenix Technologies, L. *System BIOS for IBM PCs, Compatiables, and EISA Computers*, 2nd ed. Addison Wesley, 1991.

[18] Pozzo, M. M., and Gray, T. E. A model for the containment of computer viruses. In *1989 IEEE Symposium on Security and Privacy* (1989), IEEE, pp. 312–318.

[19] RSA Data Security, I. Bsafe 3.0 benchmarks. RSA Data Security Engineering Report, 1996. http://www.rsa.com/rsa/developers/bench.htm.

[20] Schroeder, M. Engineering a security kernel for multics. In *Fifth Symposium on Operating Systems Principles* (November 1975), pp. 125–132.

[21] Tygar, J., and Yee, B. Dyad: A system for using physically secure coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.

[22] Verisign, I. Verisign certification practice statement. Tech. Rep. Version 1.1, Verisign, Inc., Mountain View, CA., August 1996.

[23] Yee, B. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.