CITI Technical Report 01-8

Personal Secure Booting

Naomaru Itoi
Center for Information Technology Integration
http://www.citi.umich.edu/
William A. Arbaugh
waa@cs.umd.edu
Department of Computer Science
University of Maryland, College Park
Samuela J. Pollack, Daniel M. Reeves
pollack@engin.umich.edu, dreeves@eecs.umich.edu
Electrical Engineering and Computer Science Department
University of Michigan

Abstract

With the majority of security breaches coming from inside of organizations, and with the number of public computing sites, where users do not know the system administrators, increasing, it is dangerous to blindly trust system administrators to manage computers appropriately. However, most current security systems are vulnerable to malicious software modification by administrators. To solve this problem, we have developed a system called sAEGIS, which embraces a smartcard as personal secure storage for computer component hashes, and uses the hashes in a secure booting process to ensure the integrity of the computer components.

May 14, 2001

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

1 Introduction

With the rapid integration of information technology into society, the demand for computer system security is soaring. Despite decades of extensive research on information security, computer systems remain vulnerable to malicious modifications. This trend reflects prevalent, but inaccurate, assumptions about computer systems: that they are trustworthy. For the purpose of this paper, we define a trusted computer as "a computer system that behaves as its users intend, without damaging or leaking the resource or information".[1] A major problem today is that modern commodity computers are not trustworthy because (1) they tend to overlook or ignore physical security issues, and (2) they are vulnerable to the exploitation of software bugs. Once an adversary compromises a computer by one of the above two methods, he can install a malicious modification that defeats any security mechanism on the computer. For example, consider a Kerberos client that steals a user's password [4], an SSL client that leaks plain text packets [5, 16], or a loadable kernel module that redirects system calls to fool a system integrity checker [1, 10].

Conventionally, the problem of trusted computing has been tackled by approaches such as access control mechanisms [6], layered architecture [17], sandboxing [9, 23], and application-level integrity checking [12]. However, all of these approaches trust the underlying hardware and operating system kernels, and are of little use if any of these components are compromised. Furthermore, many of the approaches require custom operating systems, which increases management and operational problems.

To counter this problem, Arbaugh et al. have developed a high assurance bootstrap process called *AEGIS* [2, 3]. AEGIS ensures that a valid and authorized operating system kernel is started by verifying the integrity and authorization of every component that comprises the bootstrap process through the use of digital signatures and authenticity certificates. When it boots an operating system, AEGIS guarantees that the boot process takes a valid path (in terms of integrity and authorization) from the initial power-on event to the login prompt through an inductive process [2].

Although AEGIS significantly improves the security of personal computers, it has drawbacks. First, users must trust their system administrator to authorize, i.e., digitally sign, the trusted operating systems and applications. However, because (1) security threats often come from inside of organizations, and (2) in public computing sites, such as Internet cafes and libraries, system administrators are unknown, the user may choose not to trust the administrators. Second, AEGIS is inflexible: it is difficult to change the hardware configuration of a host, and it can boot only FreeBSD.

To solve these problems, we have developed *sAEGIS*, which integrates a smartcard into the bootstrap process. In sAEGIS, we use a smartcard to store the set of component hashes that the holder of the smartcard authorizes, pushing control over the selection of approved components from the system administrator to the user. We also have ported AEGIS to support GRUB [8], a free and flexible boot loader, which supports a larger set of operating systems.

The remainder of the paper is structured as follows. First, we provide a brief review of AEGIS. Next, we present the design of sAEGIS and analyze its security. Then, we describe the implementation and provide performance benchmarks for sAEGIS. Finally, we conclude the paper and provide details of our future work.

[1] This definition is narrower than the one used in the US *Trusted Computer Security Evaluation Criteria* [19], in which the word "trusted" includes access control, covert channel analysis, etc. Our definition is closer to the ones used by Neumann [18], Brewer et al. [5], Goldberg et al. [9], and Loscocco et al. [16]

2 Background: AEGIS Secure Bootstrap Process

Here we review AEGIS to provide background for understanding sAEGIS.

AEGIS is a secure bootstrap process, whose goal is to provide a trusted foundation on a computer system. As described in Section 1, a modern computer system cannot usually be trusted because of the lack of physical security, and an untrusted initialization process. One way of addressing this problem is to ensure the *integrity* of a computer system. A system is said to possess integrity if no unauthorized modification has been made to it. Denning defines integrity similarly for communication [7]. AEGIS assures integrity of a personal computer at boot time, through a process called *chaining layered integrity checks*, which uses induction and digital certificates.

AEGIS works as follows:

1. A system administrator, or other authorized party generates a hash, H, of a bootstrap component, and creates a certificate, C, which includes a unique component identifier, an expiration date, and H.
2. The authorized party signs C with her private key.
3. C is then stored in the component if possible, and, if not, then in a data block of the flash memory device on the host's motherboard.
4. Execution control is passed to the component *if and only if:*
   (a) The certificate, C, has not expired.
   (b) The signature of C is valid, and
   (c) The hash value stored in the certificate matches the computed value of the component under consideration.

It is important to note that AEGIS provides integrity guarantees only for *starting* a system. Once a system is running, AEGIS does not provide any guarantees that the integrity of the OS remains valid.

As described in Section 1, AEGIS has two problems. First, the user is forced to trust the system administrator because the certificates are stored in the component or the BIOS, both of which are controlled by the administrator. The administrator can create and install malicious software by simply creating and signing a component certificate. AEGIS cannot detect the malicious software because the component passes all of the validity tests described earlier. The second problem is the lack of flexibility. The bootloader used in AEGIS can boot only FreeBSD. Furthermore, hardware reconfiguration on AEGIS requires the creation and installation of new device certificates. Because of its size limitation, BIOS (which is in a flash memory chip) cannot store file system drivers, and is unable to access data on the hard disk.

Readers interested in the further details of AEGIS are advised to refer to articles [2, 3].

3 Design

3.1 Design Goals

The goal of sAEGIS is as follows:

- Personalization

  In AEGIS, it is a system administrator's responsibility to manage certificates and MACs. By embracing a smartcard as a personal storage of MACs, sAEGIS hands the control to the users.

- Authentication

  In AEGIS, a user who attempts to boot a computer is not authenticated. That is, anyone who can invoke the boot process, for example, by hitting the reset button, may boot it. sAEGIS boots an operating system only if a correct smartcard and associated PIN are presented by a user. This two-factor authentication (what-you-have and what-you-know) makes theft of a mobile computer less threatening, as the thief cannot use the computer.

- Operating System Flexibility

  The only operating system the AEGIS prototype is able to boot is FreeBSD. In contrast, sAEGIS employs a free, flexible boot loader called *GRUB* [8] to boot several operating systems, namely, Linux, FreeBSD, NetBSD, OpenBSD, Windows 9*, NT, and 2000.

- Hardware Configuration Flexibility

  In AEGIS, the certificates are stored in a flash memory chip, which is hard to configure. In sAEGIS, because the smartcard access library is small enough to fit into the flash chip, the hardware configuration information, certificates, and MACs can be moved to the smartcard, which is more easily configured than the flash chip.

In the above four goals, the first three were achieved in our sAEGIS prototype. The reason why the last goal was not achieved is discussed in Section 7.

3.2 Design Overview

In a nutshell, sAEGIS = AEGIS + GRUB + smartcard + `verify`. That is, (1) sAEGIS relies on AEGIS to boot GRUB securely, (2) GRUB boots an operating system kernel securely using a smartcard for verification, and (3) the kernel checks the integrity of daemons with an application called `verify`.

The basic idea behind sAEGIS is as follows: *if a lower layer verifies the integrity of all higher layers before booting them, the system integrity is ensured.* Therefore, to comprehend the design of sAEGIS, it is essential to understand which component verifies and boots which, and how. The bootstrap process of sAEGIS is summarized in the following events, in chronological order.

1. Power on Self Test (*POST*). The processor checks itself.

   POST is invoked by either applying power to the computer, hardware reset, warm boot (*ctrl-alt-del* under DOS), or jump to the processor reset vector invoked by software. This starts the bootstrap process.

2. BIOS section 1 verifies itself and BIOS section 2, and boots section 2.

   In sAEGIS, BIOS is divided into two parts, section 1 and section 2. The former contains the bare essentials needed for integrity verification, such as a cryptographic hash function (MD5 and SHA1), a public key function (RSA), and the public key certificate of a trusted third party. The integrity of this part is assumed, i.e., it is assumed to never be modified. Discussion about this assumption is in Section 4.3.

   BIOS section 1 reads the certificate of itself from the flash chip, and verifies itself.

   BIOS section 1 reads the binary and certificate of BIOS section 2 from the flash chip, and verifies the binary. If the check goes through, it boots section 2.

3. BIOS section 2 verifies the ROM of extension cards, and executes them.

   BIOS section 2 reads the programs stored in the ROM of extension cards, reads the associated certificates from the flash chip, and verifies the programs. If the check goes though, it executes them.

4. BIOS section 2 verifies GRUB stage 1, and boots it.

   GRUB is divided into two parts, stage 1 and stage 2, because an Intel-compatible personal computer requires a primary boot loader to be no more than 512 bytes long. Stage 1 is booted by BIOS section 2; and stage 2 is booted by stage 1.

   BIOS section 2 reads the binary of GRUB stage 1 from a floppy disk, reads the certificate from the flash chip, verifies the binary, and boots it.

5. GRUB stage 1 verifies GRUB stage 2, and boots it.

   GRUB stage 1 reads the binary and certificate of GRUB stage 2 from a floppy disk, verifies the binary, and boots it.

6. GRUB stage 2 verifies the kernel and the verification tools, and boots the kernel.

   GRUB stage 2 mounts the file system (typically on a hard disk) that stores a kernel, `verify`, and a shell script that invokes `verify` (e.g., `/etc/rc.d/init.d/inet` on UNIX). It reads these files from the file system, reads the MACs from a smartcard, and verifies the files. If the check goes through, it boots the kernel.

7. The kernel uses the `verify` application to verify the important files, and starts the system daemons that pass the check.

   `verify` is invoked by the kernel at boot to check important files. If the check fails, the kernel does not start the related daemons. The important files are system daemons (e.g., `login`, `logind`, `ssh`, and `sshd` should be verified on UNIX to detect a password sniffer), configuration files (e.g., `SYSTEM.INI` should be verified on Windows to detect a Trojan horse), and shared libraries (e.g., `GINA.DLL` should be verified on Windows NT / 2000 to detect a password sniffer).

The bootstrap process is depicted in Figure 1.

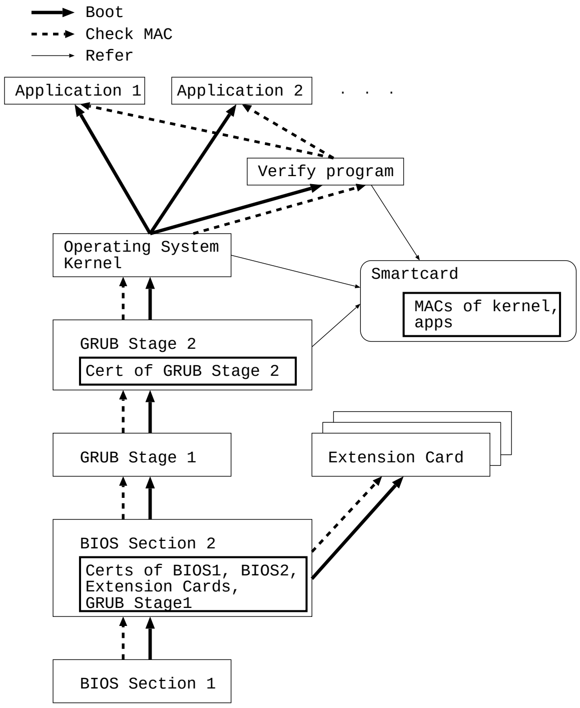

Figure 1: Bootstrap Process

3.3 Smartcard Communication Protocol

In step 6 of the list presented above, a workstation and a smartcard carry out a protocol to (1) authenticate the smartcard and (2) verify the hash presented by the workstation. The protocol is shown in Figure 2, and is described as follows.

Workstation:

- obtain PIN from the user
- compute the hash of the kernel : $m = \mathrm{SHA1}\{\mathrm{kernel}\}$
- generate a random challenge : r
- encrypt $\{m, r\}$ with public key : $\{m, r\}K_{\mathrm{pub}}$
- send $\{m, r\}K_{\mathrm{pub}}$ to the smartcard, along with the PIN

Smartcard:

- check PIN; if the PIN does not match, set ANSWER to ERR
- decrypt $\{m, r\}$ with private key
- compare m to the stored hash, and set ANSWER to OK or ERR

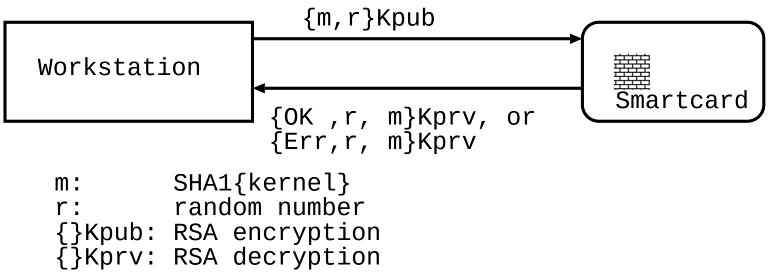

Figure 2: Smartcard - Workstation Communication Protocol

- sign $\{\text{ANSWER}, r, m\}$ with Kprv : $\{ANSWER, r, m\}K_{\text{prv}}$
- send it to the workstation

Workstation:

- encrypt $\{ANSWER, r, m\}K_{\text{prv}}$ with K_{pub}
- make sure it is signed by the smartcard.
- if (ANSWER == OK and $r ==$ original r and $m ==$ original m) continue with boot, otherwise, halt the boot process

4 Security Consideration

In this section, we discuss the security of our design.

4.1 Model

We start with constructing a model of the system. The model consists of the following participants:

Alice (A) A legitimate user who wants to boot and use a personal computer. She owns a smartcard.

Smartcard Alice's smartcard. It stores a private key, K_{prv}, and MACs. It is PIN protected, i.e., a secret number must be presented before it is used. It blocks itself if a wrong PIN is typed for n consecutive times.

Mallory (M) An adversary.

Personal Computer (PC) An Intel-compatible personal computer to be verified and booted. It consists of BIOS section 1 and 2, extension cards, GRUB boot loader stage 1 and 2, an operating system kernel, `verify`, and the other files.

4.2 Claims

Here we claim the security properties of sAEGIS.

System integrity after boot
: When a PC is booted using sAEGIS, the integrity of the following components of the PC are ensured; BIOS, extension cards, GRUB boot loader, operating system, and the other files that are verified.

User authentication
: When a PC is booted using sAEGIS, it has been booted by a legitimate user.

4.3 Assumptions

We make the following assumptions in our model.

1. BIOS section 1 is integral.

   We assume that the BIOS section 1 is not modified. This guarantees that section 1 starts up the sAEGIS bootstrap process every time the PC is booted.

   The security property of the entire sAEGIS system relies on this assumption because BIOS section 1 is the base of the secure bootstrap. If BIOS section 1 is modified maliciously, BIOS section 2 may not be verified correctly, resulting in a compromised section 2. This leads to a compromised GRUB stage 1, stage 2, and finally, a compromised operating system kernel. This defeats the goal of sAEGIS.

   We believe this assumption is reasonable. A portion of Intel's latest generation of flash ROM can be write-protected by setting one of the PINs (RP#) to high [11]. Although this protection can still be compromised by setting one of the jumper switches on a chip set, this attack can be countered by storing BIOS in ROM, prohibiting any modification.

2. Mallory can read anything in the PC, but nothing in the smartcard.

   Mallory can read any data stored in the PC. However, she cannot read any data in the smartcard. This is a reasonable assumption because it is usually easy to physically open a PC and access data storage in it. In contrast, a smartcard is tamper-resistant. While a smartcard suffers from newly developed attacks [13, 14], we ignore such attacks in this paper because (1) a smartcard is still much harder to compromise than a PC, and (2) smartcard developers are devising countermeasures to the new attacks.

3. Mallory can write anything in the PC except in BIOS section 1. She cannot write anything in the smartcard.

   Similarly to Assumption 2, Mallory can write anything in the PC except in the protected region. However, she cannot write anything in the smartcard.

4. Cryptographic functions are strong.

   We assume that cryptographic hash functions (MD5 used in BIOS, and SHA1 used in GRUB stage 2) are collision-free. We also assume that the random number generator used in the protocol given in Section 3.3 is unpredictable. Finally, we assume that our principal cipher, RSA, is impossible to compromise in a reasonable amount of time.

5. Mallory does not know Alice's private key.

6. Mallory can snoop and modify messages on the serial port in which the PC and the smartcard are communicating.

4.4 Attacks

4.4.1 Modification to PC's components

By Assumption 3, Mallory can modify anything she wants in the host except the BIOS section 1. However, if she does, Alice will notice it at the next boot because sAEGIS verifies every byte of code executed during the bootstrap process. By Assumption 1, a correct bootstrap process will be invoked every time Alice boots the PC. By Assumption 4, Mallory cannot forge a certificate or a MAC without knowing Alice's private key, and this does not happen, by Assumption 5.

4.4.2 Modification to PC's components after boot

Being a secure bootstrap system, sAEGIS makes no attempt to protect the PC after it is booted. Mallory can modify the system maliciously, e.g., install Trojan horse or a sniffer. However, Alice can always restore the integrity the PC by rebooting it.

4.4.3 Unauthorized Boot Attempt

Mallory may steal the PC and try to use it. This is impossible unless Mallory obtains Alice's smartcard and PIN, as the authentication protocol presented in Section 3.3 prevents such an attempt. Without knowing Alice's private key, K_{prv} (Assumption 2 and 5), Mallory cannot produce $\{\mathrm{OK, r, m}\}K_{\mathrm{prv}}$, because the random number generator is strong (Assumption 4).

Mallory may try to replay an OK message $\{\mathrm{OK, r, m}\}K_{\mathrm{prv}}$, but this does not work either because of the random nonce, r.

Mallory may try a man-in-the-middle attack, i.e., modifying the kernel and replacing the message from the host, $\{\mathrm{m', r}\}K_{\mathrm{pub}}$, with $\{\mathrm{m, r}\}K_{\mathrm{pub}}$. The smartcard, not knowing the hash value was altered, sends an OK message. However, the workstation notices the attack because the hash values m and m' do not match.

4.4.4 Serial Cable Wiretapping

By Assumption 6, Mallory can read and write messages on the serial cable connecting the PC and the smartcard. However, she cannot produce $\{\mathrm{OK, r}\}K_{\mathrm{prv}}$.

4.4.5 PIN Theft

Mallory may obtain Alice's PIN by breaking into the PC, or by sniffing the serial cable. This is a common problem for today's smartcard systems because a PIN is entered on the keyboard of the PC, and is transmitted to the smartcard through a serial cable. This problem can be addressed by a smartcard reader with a built-in PIN pad. For example, SPYRUS produces such a smartcard reader [22]. Another approach to this problem is to use a one-time pad for PINs, thus making replay of a PIN meaningless.

4.4.6 Mallory as System Administrator

Mallory may be Alice's malicious system administrator, and may try to compromise her secrets. For example, consider a case in which Mallory tries to read Alice's e-mail. Alice may encrypt her e-mail with a secure mail tool, e.g., PGP. However, without a system like sAEGIS, Mallory can modify the executable code of PGP to leak information. sAEGIS prevents this by detecting such modifications. If the operating system and application software vendors publish the signatures of their software, Alice can store the signatures in her smartcard, and can check the system.

It is still unclear whether we can counter all the possible attacks mounted by system administrators because security software usually is written with the assumption that system administrators are trustworthy, and attacks by system administrators have not been well studied. However, we believe that sAEGIS is the first step to counter such attacks.

5 Implementation

We describe the sAEGIS prototype, which is an implementation of the design described in Section 3. It is implemented on an ASUS P55T2P4 Pentium motherboard, running a 233 MHz AMD K6 processor.

The prototype is based on the AEGIS prototype by Arbaugh et al. We do not go into the details of the AEGIS implementation. sAEGIS uses GNU GRUB 0.5.93.1. Interested readers should consult with GRUB's website [8] for details.

5.1 GRUB stage 1

GRUB stage 1 is modified to verify GRUB stage 2 before jumping to it. Stage 1 tells AEGIS where stage 2 starts (`0x800:0`) and how large it is, and calls the AEGIS interrupt (`0xc2`).

5.2 GRUB stage 2

GRUB stage 2 is modified to carry out the protocol described in Section 3.3.

First, to communicate with a smartcard through a serial port, the smartcard communication library is implemented by replacing the system-dependent part of the sc7816 library [21] with modified serial console access routines in OpenBSD-2.4 (`/usr/src/sys/dev/ic/com.c`).

Then, it needs some cryptographic functions. SHA1 routines in GRUB are ported from Kerberos version 5-1.0.5 distributed by MIT. RSA routines are taken from PGP 2.6.2.

In this prototype, random number generation is not implemented. It is replaced with a constant.

The `kernel` command in the GRUB user interface loads a kernel from a file system to main memory. This command is modified to invoke the verification protocol before letting GRUB boot the kernel. Another command, `updatehash`, is added to update the SHA1 hash so that files can be verified in addition to the kernel.

5.3 `verify`

`verify` is a C program that reads a given file, computes its hash, verifies it with a hash stored in a file, and returns the result of verification. An example use of `verify` is as follows. In this example, `verify` makes sure `inetd` is not modified before it is started.

```
/etc/rc.d/init.d/inet:

 /boot/verify /usr/sbin/inetd
   /boot/hash-table.txt &&
   daemon /usr/sbin/inetd
```

In future implementation, `verify` should use hashes stored in a smartcard.

5.4 Smartcard-side Code

The program in the smartcard is implemented in a Schlumberger Cyberflex Access smartcard with Java. Cyberflex Access is the only smartcard we know that offers both programmability and cryptographic functions (DES, RSA, and SHA1).

The smartcard reads 128 byte input from GRUB, decrypts it with the RSA private key. It then compares the hash value with the one previously stored in its memory and determines whether the kernel image is unmodified. It concatenates its reply (0x8080808080808080 if OK, 0x4040404040404040 if not) with the random key and signs the resulting string with the RSA private key. Finally, it sends the result to GRUB.

In this prototype, the kernel hash is not included in the message sent from the smartcard to the host because the necessity for checking this value was identified after the prototype was implemented. In addition, a smartcard can hold only one SHA1 hash value. This should be improved to allow more flexibility.

6 Performance Evaluation

To evaluate the efficiency of sAEGIS, the boot process is timed. The following is the amount of time elapsed from the time that a PC is powered up until an operating system starts the last system daemon. In addition, the smartcard access time (the time spent in the protocol in Section 3.3) is measured, as it is one of the most expensive components.

Measurement was carried out on Linux 2.2 (RedHat 6.2) with a 233 MHz AMD K6 processor. We used the RDTSC instruction to obtain the number of ticks after the processor powers up. All the numbers are in seconds, and are averages of 5 trials each. Variance is small.

	time (sec)
boot with sAEGIS	69.55
boot without sAEGIS	57.88
difference	11.67

	time (sec)
smartcard access	5.54

The result shows that sAEGIS adds 11.67 seconds to the bootstrap process. About half of the added cost is for accessing the smartcard. The other half includes the following:

- Code checking, which involves MD5 hashing and RSA operations. More details about this are available [2].
- Loading GRUB, which is 77KB, from a floppy disk, takes more time than loading the much smaller (4.5KB) Linux boot loader, LILO, from a hard disk.

Adding 11.67 seconds to the bootstrap process, which already takes 1 minute, is acceptable in many environments.

7 Discussion

7.1 Key Management

To use sAEGIS effectively, it is essential to manage the private key in the smartcard appropriately. We describe two ways of managing private keys.

First, if the computer to be protected is personal, e.g., a laptop computer, one computer is associated with one owner. Therefore, the private key should be unique, and should be known only to the owner of the computer (i.e., should be only in the owner's smartcard). sAEGIS can prevent an adversary from booting the computer, thus discouraging theft of the computer. This approach may cause a problem when a smartcard is lost, broken, or stolen because the associated computer is no longer usable. Some kind of key escrow system is needed to address this problem.

Second, if the computer to be protected is public, e.g., in a library, or in an Internet cafe, one computer is associated with many users. The current sAEGIS prototype cannot provide such multi-user authentication because it has only one key pair between the smartcard and the computer. To achieve this, some multi-user authentication mechanism is necessary, e.g., a certificate-based mechanism with revocation, or a symmetric key-based mechanism such as Kerberos. An alternative to this is not to authenticate users at boot time, let anyone boot the computer, and rely on application level authentication. sAEGIS can achieve this by assigning the same private key for multiple users. The trade-offs between these two approaches are under discussion.

7.2 Future Direction

7.2.1 Fix Implementation Limitations

Four implementation limitations described in Section 5 should be fixed, namely, (1) no random number generator, (2) `verify` does not use a hash in the smartcard, (3) kernel hash, m, is not included in the message the smartcard sends to the workstation, and (4) the smartcard holds only one hash.

7.2.2 Smartcard Access from BIOS

To achieve Goal 4 described in Section 3.1, it is necessary to move the smartcard access library into BIOS. The library is 11 KB, so the size should not be a problem for the 1M flash BIOS.

Unfortunately, one of the authors who was responsible for smartcard programming did not have permission to access the BIOS source code. Instead of working out licensing issues, we decided to implement a prototype, and to wait until open-source BIOS projects are mature enough to be used as the next platform [15, 20].

8 Conclusion

We have implemented a personal, secure bootstrap process, sAEGIS, which is an extension to AEGIS. Advantages of sAEGIS over AEGIS are: (1) the smartcard lets users control what they use, (2) the smartcard serves as an authentication token, and (3) it is more flexible than AEGIS.

The following two aspects highlight the value of this work.

- Improvement to important software

  As attacks that modify an operating system itself are becoming more common, secure bootstrap, such as AEGIS, is strongly demanded. One of the problems of AEGIS is the lack of flexibility: it can only boot the FreeBSD kernel, and it requires reprogramming of a flash chip when the hardware configuration is changed. We solved the former problem, and proposed a solution to the latter.

- Idea of personalization

  sAEGIS suggests a system in which the user does not have to trust system administrators. We believe it is a huge security gain because many attacks come from inside organizations.

Acknowledgments

We thank Jim Rees at the University of Michigan for directing us about serial communication in a boot loader. We thank Professor Peter Honeyman and Professor Brian Noble at the University of Michigan for their advice.

This work was partially supported by a research grant from Schlumberger, Inc.

References

[1] Rootkit homepage. http:// www.rootkit.com/.

[2] William A. Arbaugh. *Chaining Layered Integrity Checks*. PhD thesis, University of Pennsylvania, 1999.

[3] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[4] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *Proceedings of the Winter 1991 Usenix Conference*, January 1991. ftp://research.att.com/dist/internet_security/ kerblimit.usenix.ps.

[5] Eric Brewer, Paul Gauthier, Ian Goldberg, and David Wagner. Basic flaws in internet security and commerce, 1995. http:// www.ao.net/ netnigga/ endpoint-security.html.

[6] A. Dearle, R. di, J. Farrow, F. Henskens, D. Hulse, A. Lindstrm, S. Norris, J. Rosenberg, and F. Vaughan. Protection in the grasshopper operating system, 1994.

[7] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, 1983.

[8] Free Software Foundation. Gnu grub, 1999. http://www.gnu.org/ software/ grub/ grub.html.

[9] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications. In *Proceedings of 6th USENIX Unix Security Symposium*, July 1996.

[10] halflife. Bypassing integrity checking systems. Phrack Magazine, September 1997. Volume 7, Issue 51, Article 9 of 17.

[11] Peter Hazen. Flash memory boot block architecture for safe firmware updates. Technical Report AB-57, Intel, 1995. http://developer.intel.com/ design/flcomp/ applnots/292130.htm.

[12] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. Technical report, Purdue University, 1995. CSD-TR-93-071.

[13] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Introduction to differential power analysis and related attacks. Cryptography Research, 1998. http://www.cryptography.com / dpa / technical / index.html.

[14] Oliver Kommerling and Markus G. Kuhn. Design principles for tamper-resistant smartcard processors. In *Proceedings of USENIX Workshop on Smartcard Technology*, Chicago, May 1999.

[15] Linux bios. http://www.acl.lanl.gov/ linuxbios/.

[16] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference*, Crystal City, Virginia, October 1998. National Security Agency, NISSC. http://www.jya.com / paperF1.htm.

[17] H. Nag, R. Gotfried, D. Greenberg, C. Kim, B. Maccabe, T. Stallcup, G. Ladd, L. Shuler, S. Wheat, and D. van Dresser. Prose: Parallel real-time operating system for secure environments, 1996.

[18] Peter G. Neumann. Architectures and formal representations for secure systems, 1996. Technical Report SRI-CSL-96-05, Computer Science Laboratory, SRI International.

[19] Department of Defense. Trusted computer system evaluation criteria, December 1985. http:// www.radium.ncsc.mil/ tpep/library/ rainbow/5200.28-STD.html.

[20] Openbios. http://www.freiburg.linux.de/ OpenBIOS/.

[21] Jim Rees. Iso 7816 library, 1997. http://www.citi.umich.edu / projects / sinciti / smartcard / sc7816.html.

[22] Spyrus. http:// www.spyrus.com/.

[23] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. client software-based fault isolation, 1993.