Proceedings of the 2006 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY, 21-23 June 2006

Toward a Boot Odometer

Richard C. Vernon, Cynthia E. Irvine, Timothy E. Levin

Abstract—

In trustworthy systems, object reuse requirements extend to all forms of memory on the platform and can include volatile elements such as RAM, cache, I/O device registers, and certain controllers. To ensure that residual information is not accessible from one session to another, these regions must be either protected or purged. In situations where the operating system cannot be trusted to meet object reuse requirements, an alternative is needed.

In this paper, we address the object reuse problem in volatile memory. A "hard" reboot includes a power cycle, which ensures that sensitive information in volatile memory is purged, whereas a software initiated reboot does not. How can we prove that a hard reboot has occurred? To our knowledge, it is not possible for a remote entity using currently available technology, to sense whether a hard reboot has occurred on an PC client, e.g. between communication sessions. We propose a hardware-assisted design that uses a secure coprocessor to sense the reboot type of the host platform and that maintains a Boot Odometer that tracks the sum of hard reboots that have occurred on the host. In addition, secure coprocessor services allow trustworthy attestation to a remote entity, cognizant of a previous Boot Odometer Value, that volatile memory has been purged.

I. Introduction

The limited nature of the physical resources in computers requires management of their allocation so that the same physical resources can be reused for objects to which different security attributes may be assigned. A form of passive misuse of computers is the scavenging of pre-existing information on storage resources that have been allocated to a new object [1]. This *object reuse problem* must be addressed in systems enforcing access control and information flow policies.

In general, to address object reuse, residual information is purged from resources (e.g., physical memory) between the time they are deleted from one object and reallocated to another object [2]. The Common Criteria devotes an entire functional family (i.e., FPT-RIP, Residual Information Protection) [3] to object reuse.[1]

A remote content server or users of a public computer at a library or coffee shop could need to ensure that information from one session does not persist to the next session. If the operating system cannot be relied upon to address object reuse to an adequate level of trustworthiness, are there effective alternatives to doing nothing?

We have been engaged in the development of a distributed client-server architecture (MYSEA) that segregates highly sensitive information from less sensitive information, while providing assured sharing as permitted by an overarching system policy. [4], [5]. The architecture leverages a combination of trusted systems and commercial products to implement multilevel networks over which secure information services can be provided. To achieve user acceptability [4], viz. usable security, the architecture allows client systems to execute any operating system and any suite of application software and are assigned a current session level, which may change over time. By definition, the clients may not be trustworthy and may even include malicious code in the OS. It is assumed that users are not a threat to the physical integrity of the client systems. We can achieve this by trusting the users, guarding the client, or using tamper-proof techniques. As will be seen, the tamper resistance properties of the Trusted Platform Module [6] determine the integrity of our prototype. The only constraint on clients is that all session-related information must be stored on a highly trusted server. Thus the clients have large RAM disks, but no persistent writable storage. When the client changes session level, it has access to different information on the server. Care must be taken to ensure that information from previous, more sensitive sessions is not accessible in local volatile memory devices on the client.

A system *reset*, or manual power down of the client, causes the refresh rate for memory to go to zero. As a result, information in volatile memory is zero and when the system is restarted, memory will contain either random or some predefined constant values. The operating system can also be restarted without a powercycle. Here we call this a *soft reboot* (also called a *warm reboot*). A reboot that involves a power cycle is called a *hard reboot*. In the case of a soft reboot, the problem of residual information persists regardless of the boot medium: CD, USB drive, etc.

An earlier study of object reuse in the context of potentially malicious client operating systems [7] concluded that power cycling the hardware was the best approach for purging volatile memory. Without a power cycle on reboot, sensitive information may remain in volatile memory

R.C. Vernon: Naval Postgraduate School, Monterey, CA.
C. E. Irvine: Naval Postgraduate School, Monterey, CA.
T. E. Levin: Naval Postgraduate School, Monterey, CA.

[1]Note that these requirements do not address hardware forensics.

ISBN 0-7803-9814-9/$10.00

components, including system RAM and other board-level hardware components. However, at that time there appeared to be no foolproof way to validate that a power cycle had occurred.

A trusted coprocessor [8], [9] offers a path to a solution. A secure coprocessor added to the main system board of the client can provide a trustworthy environment for the storage of security-critical information and for performing security-critical computation. Its security and encryption services provide a context in which a *hard* reboot of the client can be attested.

The problem of trusted system boot and the use of trusted coprocessors to facilitate this process has been discussed in earlier work [10], [11], [9]; however, this earlier work did not address the problem of object reuse as it relates to ensuring that the system has undergone a power cycle.

This paper presents a set of enhancements to the TPM specification [8] that would allow it to sense the type of reboot a computer has undergone: hard or soft. We show that the proposed design reflects key security properties [1]: that it is understandable, cannot be bypassed, and cannot be modified. The TPM is also enhanced to maintain a *Boot Odometer Value* (BOV), which tracks the number of hard reboots the host platform has undergone. When used in conjunction with a TPM that provides attestation services[2], the platform can trustfully prove to a remote entity, such as our trusted server, the current total number of hard reboots and the type of the most recent reboot. To further explore our ideas, a proof of concept hardware simulation was developed.

II. Background

Most modern, networked applications implicitly trust the integrity of the systems with which they communicate. However, even if a software application is trustworthy, there is no way to verify the integrity of the operating environment of today's commercial systems. When queried, malicious software can lie with impunity - and malicious applications can often exploit weaknesses in commercial operating systems.

Secure coprocessors can be used to provide high integrity evidence of the client's current operating environment, such as hardware, operating systems, and running software, to a remote entity. This information can establish a basis for decisions by remote entities regarding the provision of services and data to the attesting system. The Trusted Platform Module [8] can be used to provide a trustworthy source of platform measurements. Figure 1 presents a generic representation of a TPM-enhanced architecture.

The security architecture in which we have explored client object reuse issues consists of a completely untrusted client, a small trusted appliance called a Trusted Path Extension (TPE), and a trusted server, which manages the MYSEA LAN [4]. Both the TPE and the server are high assurance components and constitute elements in the distributed trusted computing base (TCB). The TPE provides a user with trusted I/O and protects itself from malicious software operating on the client computer through its physically separate execution environment and its high assurance separation kernel [12]. TPE services include:
1. a high assurance trusted path [13] to the server for user identification, authentication and session security attribute negotiation,
2. support for a protected communication channel between itself and the server,
3. trusted screen and keyboard interfaces,
4. dynamic security services such as context and application-specific security associations [**?**], and
5. mediation of client access to the LAN.

The concept of operation for client access to server resources requires that all security-relevant parameters for a user session must be established to the server using the TPE before the untrusted client can access the network. If the client has requested a change to a lower session level, volatile memory must be purged on the client. While the TPE can provide security services directly between the user and the trusted server, neither the TPE nor the trusted server can currently validate a hard reboot which implies memory purge of the client computer.

To address this problem, a secure coprocessor, viz., a modified TPM, has been added to the TCB to detect the type of reboot the client platform underwent, and to keep a count of the number of client platform hard reboots. An instruction is added to the TPM that allows local programs to query for the hard reboot count and provide that trustworthy evidence to the trusted server.

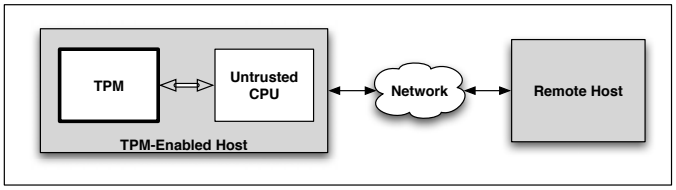

Fig. 1. Simple Layout for TPM-enhanced Operations.

III. Design

Each client system is equipped with a TPM, i.e., a secure coprocessor, as illustrated in Figure 2. The secure coprocessor stores a high integrity number known as the Boot Odometer Value (BOV). If and only if the computer hard reboots, the secure coprocessor increments the BOV by 1.

Using the TPM's attestation service, the client can prove to a remote entity, that is cognizant of a previous BOV, that volatile memory has been purged.

[2] "Attestation is the process of vouching for the accuracy of information" [8]

ISBN 0-7803-9814-9/$10.00

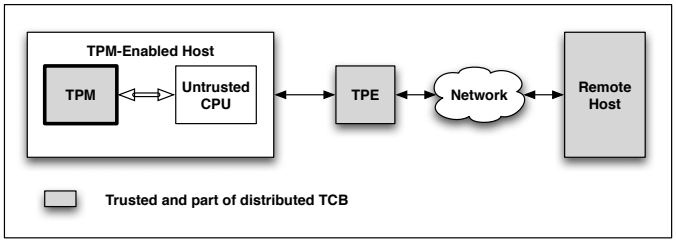

Fig. 2. Client Enhanced with TPE and TPM.

To do this, the client requests its secure coprocessor to sign the current BOV using a private key known only to the secure coprocessor.[3]The client then sends the signed BOV to the remote entity. The remote entity uses the corresponding public key to validate the BOV. Assuming that the secure coprocessor protects its private signing keys from tamper or observation, the remote entity can trust that the BOV is authentic and has not been altered in transport. By comparing the current BOV to previously provided BOVs, the remote entity can determine if an additional hard reboot has occurred on the client. Protection against replay and other network attacks is provided by the attestation protocol [14].

A. Highlevel System Requirements

The Boot Odometer function results in several high-level design requirements.

- The BOV must increment every time a full power cycle-boot sequence completes. No requirement is made regarding the value of the increment; however, an increment of 1 is preferred to minimize the number of rollover events in a finite storage location.
- The BOV must only increment on a hard reboot. (It must not change during a soft reboot.)
- The BOV must be retained during system power-off state: it must be located in non-volatile storage.
- The BOV must be protected from software tampering from other elements on the host. Access to the secure coprocessor services is negotiated via software calls. It must not be possible to use any combination of software calls to the secure coprocessor to change the BOV.
- The BOV increment operation must be functionally atomic. The BOV must not be corruptible by power cycles initiated during the boot sequence, i.e., increment without a hard boot.
- The secure coprocessor must be able to detect error conditions in the Boot Odometer mechanism. Because the Boot Odometer mechanism is part of the system boot routine, errors in its operation can affect system security by corrupting the boot process. By detecting exceptions, the secure coprocessor can take appropriate action to ensure system confidentiality or integrity.
- The secure coprocessor must be able to respond to error conditions in the Boot Odometer mechanism. Exception handling must be well defined and not leave the coprocessor in an insecure state.
- The secure coprocessor must be able to attest to the value of the BOV with high integrity.

The Boot Odometer operates as follows. One of the TPM Platform Configuration Registers (PCR) is used to hold a *Boot Status Indicator* (BSI). If the platform has undergone a hard reboot, power to the TPM will have been interrupted causing its PCRs to initialize to a known value [6]. For simplicity, we assume that the initialization value is all zeros. A soft reboot will not reset the contents of the PCRs. During its initialization phase, the TPM inspects the PCR where it had previously placed the Boot Status Indicator. If the Indicator appears inside the specified PCR, rather than the initialization value, then the TPM assumes that it has undergone a soft reboot and the TPM is disabled. (Note that this rather Draconian approach is sufficient for the demonstration of concept of the goals of the security architecture and will be the focus of refinement in subsequent work. For example, instead of disabling the TPM the BSI could be used to indicate the type of the boot: hard or soft.) Disabling the TPM does not prevent the host platform from accessing the hardware accelerated encryption algorithms provided by the TPM [8], it does prevent the host platform from accessing services that are associated with TPM protected encryption keys.

In the case of a hard reboot, the TPM first updates the contents of the specified PCR with the Boot Status Indicator, then it fetches the Boot Odometer Value from non-volatile TPM storage, increments it, and finally places the BOV back into the non-volatile storage of the TPM. The initial value of the Boot Odometer Value is assumed to be set at the factory, but has no relevance to the correctness of the BOV algorithm. The non-volatile memory and the PCR in which the Boot Status Indicator is stored are not accessible by the host platform CPU. This protects them against malicious alteration.

A remote entity has two choices when requesting trusted attestation of a hard reboot. First, the remote entity can store a previously provided BOV. It can then request the current BOV. If the current value is larger than the stored value, then the remote entity can assume that the client has undergone a hard reboot. Second, it can ask the host to directly attest to the fact that the current session is the result of a hard boot. In this case, the client would ask its TPM, if it is enabled, to return a time-stamped and signed record that can be sent as evidence to the remote entity; (see Section III-C, Software, for details). If the session is the result of a soft boot, such a record cannot be produced.

[3]This is a highly simplified description, the details of which are elsewhere, e.g., [14].

B. Hardware

The Boot Odometer mechanism is implemented by making changes to the initialization firmware within the TPM [8]. The changes required for the BOV will affect the TPM_Startup function, which executes every time the TPM goes through an initialization cycle. TPM_startup is well suited for incorporation of the Boot Odometer functionality.

The TPM_Startup function call is part of the TPM initialization phase. At the beginning of every boot cycle, the TPM undergoes a transition function called TPM_Init which transitions the TPM into its first stage of initialization and behaves identically whether the system underwent a hard or soft reboot. TPM_Init places the TPM in a state where it waits for an external command to execute TPM_Startup. Platform initialization code must inform the TPM what type of initialization it is currently undergoing. The TPM_Startup function behaves differently based on one of three flags. The TPM_ST_CLEAR flag signals the TPM to reset volatile TPM variables back to their default values. The TPM_ST_SAVE flag signals the TPM to restore volatile variables back to their previously known values. This occurs when the computer starts from a hibernation related state. The TPM_ST_DEACTIVATED flag signals the TPM to enter a deactivated state.[15] Because we are only interested in the situation were the computer boots into fully operational mode from a power-off state, only the case were TPM_Startup is called with the TPM_ST_CLEAR flag is considered.

The TPM Specification requires that all system boots must first start with a system-wide reset. This includes physically signaling system components that a system boot is happening. This requirement prevents the TPM from being maliciously reset without a platform-wide reset, a situation that would render the TPM vulnerable to certain masquerade attacks.[15]

The required actions taken by a Boot Odometer-enhanced TPM when executing TPM_Startup are presented in Table I in standard TPM specification format.[15] Steps marked in bold indicate changes made to accommodate the Boot Odometer mechanism. As noted, a result of a hard reboot is that all PCRs are set to known initial values.

The test for the Boot Status Indicator (BSI) occurs at *step a* and before the PCRs are reset (*step c*). PCR number 8, hereafter abbreviated PCR[8], is used as the container for the Boot Status Indicator. Because PCR[8] is in volatile memory, its contents will already have been cleared when there is a power cycle reboot. The Boot Status Indicator is an arbitrary, but constant, binary string that is different from the PCR initial value and is no larger than a PCR. If PCR[8] equals the Boot Status Indicator, it indicates that the computer did not undergo a hard reboot, the TPM is disabled and can not be re-enabled until the computer is hard rebooted.

Absence of the BSI indicates a hard reboot [16]. [4] Since the contents of PCR[8] are reset at *step c*, the Boot Status Indicator is not re-entered into PCR[8] until *step d*. At *step e* the BOV is incremented to keep track of the number hard reboots. No count is kept for the number of soft reboots. The BOV must be stored in TPM non-volatile storage such that its integrity is assured by the TPM. The tamper resistance properties of the TPM [6] determine the integrity of the BOV.

C. Software

If the computer is running in a TPM-enabled state, any program that has access to the TPM can determine that the computer session was started with a hard reboot when it receives a response to TPM key-based function (see Section III-A). However, the converse is not true. A TPM maybe disabled for several reasons. A program cannot assume that a disabled TPM indicates a soft reboot.

To obtain the current BOV, software on the host will need to query the TPM using a new instruction that returns the current BOV, such that it is cryptographically signed by the TPM if requested.

To compute the boot status of the client, the remote entity must perform three functions. First, it must be able to cryptographically validate an attestation response from a client. Second, it must be able to securely store validated BOVs. High assurance systems could do this easily, whereas less trusted systems equipped with TPMs could utilize the TPM Sealed Storage service. Third, the remote entity must be able to reliably compare the currently attested BOV with the previously attested BOV.

Servers that retain previously attested client BOVs must be able to handle BOV rollover events. These will be instantly recognizable since the current BOV will be smaller than the previously observered BOV. In this situation, the server must make a decision to either accept or not accept the new BOV. This decision could be based on whether or not the wrap-around difference between the previoius BOV and the current one is greater than some preconfigured value. For example, if the previous BOV was the maximum possible BOV value, and the current BOV is 0 or a small integer, then this could be seen as correct operation of the client system.

IV. Hardware Prototype

A hardware simulation was generated using Version 3 of SimpleScalar and was based on the SimpleScalar/PISA target architecture.[17], [18], [19] The PISA target was assumed to be a generic representation of the TPM module

[4] Earlier versions of the TPM specification did not require initialization of TPM memory to known values. In this case, there is a possibility of a collision with the BSI. For an n-bit register, the probability that the BSI would be found in PCR[8] is 2^{-n}.

TABLE I
ACTIONS OF BOOT ODOMETER-ENHANCED TPM

Steps in boldface type are modifications to the standard TPM specification [15] that permit the Boot Odometer mechanism.

1. If stType = TPM_ST_CLEAR

(a) Inspect the contents of PCR[8].
i. **If the Boot Status Indicator is found, disable the TPM.**
(For example, the TPM_ST_DEACTIVATED flag might be set and then there might be a jump to the code for stType=TPM_ST_DEACTIVATED.)

(b) Ensure that sessions associated with resources TPM_RT_CONTEXT, TPM_RT_AUTH and TPM_RT_TRANS are invalidated.

(c) Reset each PCR value to its default value (reworded from TPM specification for clarity)

(d) **Set the contents of PCR[8] to the Boot Status Indicator.**

(e) **Increment the Boot Odometer Value by 1 by doing the following:**
i. Read the current BOV from non-volatile storage
ii. Increment the BOV by 1 using the TPM processor
iii. Write the new BOV back to non-volatile storage

(f) Set the following TPM_STCLEAR_FLAGS to their default state.
i. Physical Presence
ii. PhysicalPresenceLock
iii. disableForceClear

(g) The TPM MAY initialize auditDigest to NULL
i. If not initialized to NULL the TPM SHALL ensure that auditDigest contains a valid value
ii. If initialization fails the TPM SHALL set auditDigest to NULL and SHALL set the internal TPM state so that the TPM returns TPM _FAILED_SELFTEST to all subsequent commands.

(h) The TPM SHALL set TPM_STCLEAR_FLAGS → deactivated to the same state as TPM_PERMANENT_FLAGS → deactivated.

(i) The TPM MUST set the TPM_STANY_DATA fields as follows:
i. TPM_STANY_DATA→contextNonceSession is set to NULLS
ii. TPM_STANY_DATA→contextCount is set to 0
iii. TPM_STANY_DATA→contextList is set to 0

(j) The TPM MUST set TPM_STCLEAR_DATA fields as follows:
i. Invalidate contextNonceKey
ii. countID to NULL
iii. bGlobalLock to FALSE

(k) Determine which keys should remain in the TPM

(l) For each key that has a valid preserved value in the TPM
i. if parentPCRStatus is TRUE then call TPM_FlushSpecific(keyHandle)
ii. if IsVolatile is TRUE then call TPM_FlushSpecifid(keyHandle)

and was assumed to logically simulate the general-purpose computation carried out within the TPM.

The simulation indicates the feasibility of the TPM enhancements to accommodate the Boot Odometer mechanism. A combination of both interactive and scripted testing showed that the mechanism maintained the BOV in protected store and performed register operations, including register rollover, correctly.Because the mechanism is needed only infrequently, its performance is not critical. For the prototype, no performance measurements were conducted.

V. SECURITY ANALYSIS

The non-volatile memory and the PCR in which the BSI and the BOV are located are not accessible by the host platform CPU. This protects them against malicious alteration.

The order of steps c, d, and e in TPM_Startup described in Section 3 is such that interruption of the start up sequence could, at worst, result in a false negative, i.e. indication that the system did not undergo a hard reboot. However, it is expected that if an external interrupt occurred during this stage, the system would require a hard

reboot to achieve a known state.

It is not possible to exploit the BOV as a useful covert channel because the BOV only increments in cases where the computer is hard rebooted and software in current PC architectures cannot initiate a hard reboot, i.e., the best they can do is "shut down". Manually modulated covert channels are of no interest since a user-initiated channel is outside of the scope of the technical security policy. Nevertheless, while the computer could be used to *assist* the malicious user in the exfiltration of data, only a single bit of sensitive information is transmitted per reboot. Assuming a relatively fast boot process of 15 seconds, the channel rate would be only four bits per minute, which is well below the threshold for concern [13].

There is no need for the instruction that returns the current BOV to be privileged. First, there are no software-driven covert channels to be exploited in conjunction with the BOV. Second, the TPM cannot be modified by software.

The standard protocols associated with the attestation process ensure that replay attacks are infeasible [14]. A malicious client could not "store up" BOVs and at some point in the future attempt to trick the server into believing that a hard boot had just occurred.

Because the function calls for accessing the BOV and initializing the TPM are accessible to untrusted components, a malicious user might attempt a denial of service attack on the TPM by flooding it with requests to perform the function. This is the same for all non-privileged TPM calls. The Trusted Software Stack (TSS) mitigates such denial of service attacks [20]. However, a malicious BIOS could cause a denial of service by passing the ST_DEACTIVATED flag to the TPM at startup.

VI. Related Work

The problem of object reuse on commodity PCs was investigated by Agacayak, who identified areas on the platform that could possibly contain residual information following a warm boot [7]. His conclusion that a power cycle will clear dynamic memory areas (including main memory) is the basis for using a hard reboot to address the object reuse problem. Both Agacayak and Turin noted the problem of ensuring a power cycle reboot from an associated secure coprocessor [7], [21].

Secure boot processes [10], [11], [9] are based on the fact that system initialization begins at a specific point in program code. The boot process then proceeds in a layered fashion such that each layer verifies the integrity of the layer above it before passing control. None of these approaches attempts to distinguish a hard reboot from a soft reboot; therefore, none are able to address the object reuse problem on untrustworthy clients.

Smith utilizes a register with a non-negative value, termed the *trust ratchet*, within the secure coprocessor to keep track of the layer at which it is currently executing. Each boot layer increments the trust ratchet before passing control to the next layer [9]. The focus is the integrity of the system software, not a guarantee that all volatile memory on the platform has been purged since the previous session. In addition, the boot processes described in these other efforts make no use of a long term stored secret that could be compared to the boot odometer.The BirliX architecture does not address the distinction between soft and hard system boots [22], [23]. Among its goals are attestation and the use of certificates and stored secrets for that purpose. Integrity of the bootstrap is listed among its objectives, but it is silent regarding the distinction between hard and soft boot [23]. The Aegis system [11] did not employ a secure coprocessor, but still depended on a trusted BIOS whose integrity was explicitly trusted. Aegis does not address object reuse for the platform, but is focused on the integrity of the operating system. In addition, the Aegis boot process explicitly permits warm boot.

VII. Discussion and Summary

A TPM-based solution for the platform-level object reuse problem assumes a larger infrastructure in support of a distributed TPM architecture, and includes TPM equipped machines, TPM aware operating systems, and TPM aware applications. It is believed that the investment in the TPM by hardware and software manufacturers ensures that expanded support for the TPM is forthcoming.

In addition to the object reuse problem, there are several other uses for the Boot Odometer Value, if the mechanism is extended to separately log soft as well as hard reboots. In a corporate network, policy may dictate that computers should only be used for work purposes and that no other software should be installed or used. If some users are using bootable CDs to boot into other operating systems and bypass client-enforced policy settings, this could be detected by observing that the soft or hard Boot Odometer Values have been incremented more than expected between concurrent attestations, although this is not conclusive evidence that policy has been violated. The BOV could also be incremented if the system is undergoing reboots because of a malfunction; however, either situation would warrant further investigation.

Sometimes it is necessary to ensure that a host has undergone a reboot regardless of its type. For example, following the application of security patches to system software, it is often the case that a system must undergo a reboot. A remote observer who needs to ensure compliance with this procedure could request attestation that a reboot has occurred.

Managers of large data centers could also use the BOV. Suppose a data center loses partial power. Machines may be rebooted, but need some manual intervention to recover to full operational status. A system administrator could

request that all remote hosts attest to their current Boot Odometer Values. Comparison of current and previous BOVs would allow the system administrator to determine which machines had lost power and needed further attention.

A. Summary

We described the design of a hardware-enabled mechanism to sense a hard reboot through the use of a modified Trusted Platform Module. Analysis has shown the conceptual design to be secure because it is understandable, cannot be bypassed, and cannot by modified. The design can be considered understandable because of its simplicity. It included only four new requirements to the TPM_Startup function. The new operations cannot be bypassed because they are part of TPM function TPM_Startup. This function occurs during every system boot cycle and is specified in TPM design documents. The added functionality is non-modifiable because it is protected by the TPM.

Using a full specification of a commercially produced TPM, further study on the integration of the Boot Odometer concept into a full TPM could be conducted. Foreshadowing this work, the extension of the Boot Odometer to track soft reboots was also described.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. DUE-0114018 and Grant No. CNS-0430566. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. P. Anderson, "Computer security technology planning study," Tech. Rep. ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, 1972. (Also available as Vol. I,DITCAD-758206. Vol. II, DITCAD-772806).

[2] National Computer Security Center, "A guide to understanding object reuse in trusted systems," Tech. Rep. NCSC TG-018, National Computer Security Center, Fort George G. Meade, MD, 1991.

[3] ISO/IEC, "ISO/IEC 15408 - Common Criteria for Information Technology Security Evaluation." Version 3.0, July 2005.

[4] C. E. Irvine, T. E. Levin, T. D. Nguyen, D. Shifflett, J. Khosalim, P. C. Clark, A. Wong, F. Afinidad, D. Bibighaus, and J. Sears, "Overview of a High Assurance Architecture for Distributed Multilevel Security," in *Proceedings of the 2004 IEEE Systems Man and Cybernetics Information Assurance Workshop*, (West Point, NY), pp. 38–45, June 2004.

[5] T. D. Nguyen, T. E. Levin, and C. E. Irvine, "MYSEA testbed," in *Proceedings of the 6th IEEE Systems, Man and Cybernetics Information Assurance Workshop*, (West Point, NY), pp. 438–439, June 2005.

[6] Trusted Computing Group, "TCG specific implementation specification, version 1.2," tech. rep., Trusted Computing Group, 2005.

[7] C. Agacayak, "TCBE control of object reuse in clients," Master's thesis, Naval Postgraduate School, Monterey, CA, March 2000.

[8] Trusted Computing Group, "TCG specification architecture overview," Tech. Rep. Rev 1.2, Trusted Computing Group, 28 April 2004.

[9] S. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor," *Computer Networks*, vol. 31, pp. 831–860, November 1999.

[10] B. Yee, "Using secure coprocessors," Tech. Rep. CMU-CS-94-149, Carnegie Mellon University, Pittsburg, PA, 1994.

[11] W. A. Arbaugh, D. Faber, and J. Smith, "A secure and reliable bootstrap architecture," in *Proceedings 1997 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 65–71, May 1997.

[12] T. D. Nguyen, T. E. Levin, and C. E. Irvine, "Tcx project: High assurance for secure embedded systems," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 21–25, March 2005.

[13] *Department of Defense Trusted Computer System Evaluation Criteria*. No. DoD 5200.28-STD, National Computer Security Center, December 1985.

[14] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings 11th ACM Conference on Computer and Communications Security*, (Washington, DC), pp. 132–145, ACM Press, October 2004.

[15] Trusted Computing Group, "TPM main, part 3, commands," Tech. Rep. Specification Version 1.2, Level 2 Revision 85, Trusted Computing Group, 13 February 2005.

[16] Atmel engineers, "Private communication," August 2005.

[17] T. Austin, E. Larson, and D. Ernst, "SimpleSclar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, pp. 59–67, February 2002.

[18] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0," Tech. Rep. CS-TR-97-1342, University of Wisconsin, Madison, Wisconsin, 1997.

[19] T. Austin, "Simplescalar 3.0 release." http://www.simplescalar.com/docs/ANNOUNCE-3.0d.txt, October 2003.

[20] M. F. Barret, "Towards an open trusted computing framework," Master's thesis, University of Auckland, Auckland, New Zealand, 2005.

[21] B. Turan, "Client bootstrap under tcbe control," Master's thesis, Naval Postgraduate School, Monterey, CA, March 2000.

[22] H. Hartig, O. Kowalski, and W. Kuhnhauser, "The BirliX security architecture," *Journal of Computer Security*, vol. 2, no. 1, pp. 5–21, 1993.

[23] H. Hartig, "Security architectures revisited," in *Proceedings 10th ACM SIGOPS European Workshop*, (Saint-Emilion, France), September 2002.