USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

Flexible OS Support and Applications for Trusted Computing

Tal Garfinkel Mendel Rosenblum Dan Boneh
{talg,mendel,dabo}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

Trusted computing (e.g. TCPA and Microsoft's Next-Generation Secure Computing Base) has been one of the most talked about and least understood technologies in the computing community over the past year. The capabilities trusted computing provides have the potential to radically improve the security and robustness of distributed systems. Unfortunately, the debate over its application to digital rights management has caused its significant other applications to be largely overlooked. In this paper we present a broader vision for trusted computing. We give an intuitive model for understanding the capabilities and limitations of the mechanisms provided by trusted computing. We describe a flexible OS architecture to support trusted computing. We present a range of practical applications that illustrate how trusted computing can be used to improve security and robustness in distributed systems.

1 Introduction

Many difficult problems in today's distributed systems, such as preventing denial of service, performing access control and monitoring, and achieving scalability, are either caused or severely exacerbated by the fact that clients are untrusted and thus potentially malicious. This forces system designers to implement most system policy and sensitive computations in the core of the system, where trust resides, instead of at the endpoints where most of the system's resources and capabilities are. The only complete solution to this problem has been the use of closed platforms, such as those in cellular networks and banking systems, where special-purpose, tamper-resistant clients are utilized that provide end-to-end trust. This approach has demonstrated significant benefits, allowing the construction of some of today's most capable and robust distributed systems. Unfortunately, this approach presently necessitates the use of dedicated hardware, thus limiting designers to the use of only a few types of devices over which they must have exclusive control.

In the near future it will no longer be necessary to force designers to make trade-offs between the benefits of open and closed platforms. This change will come as the result of ubiquitous support for trusted computing platforms. Trusted platforms will allow systems to extend trust to clients running on these platforms, thus providing the benefits of open platforms: wide availability, diverse hardware types, and the ability to run many applications from many mutually distrusting sources while still retaining trust in clients.

The vision of trusted platforms cannot be achieved with today's operating systems which offer poor assurance and implement a security model that is largely orthogonal to that required for trusted computing. To meet the demands of implementing a trusted platform we outline the design of a new OS architecture based on the idea of a trusted virtual machine monitor. In this model, traditional applications and OSes can run side-by-side on the same platform in either an "open box" or "closed box" execution model in keeping with the trust requirements imposed by the application.

In the next section we define and describe the components that make up trusted computing. In Section 3 we present our approach of using a trusted virtual machine monitor to support a mixture of open and closed box models simultaneously. In Section 4 we examine a selection of practical areas where trusted computing can provide novel functionality yielding significant benefits for security, scalability and robustness. Section 5 discusses related work.

2 Trusted Platforms

Open platforms are general-purpose computing platforms where there is no apriori trust established between the hardware of the platform and a third party, that could be used to prove the functionality of the platform. Examples of these include workstations, mainframes, PDAs, and PCs. Open platforms possess many practical benefits over closed platforms. Unfortunately a remote party cannot make any assumptions about how that platform will behave or misbehave.

Closed platforms are special-purpose computing devices that interact with the user via a restricted interface (e.g. automated tellers, game consoles, and satellite receivers). A closed platform can authenticate itself as an authorized platform to a remote party using a secret key embedded in the platform during manufacturing. Closed platforms rely on hardware tamper resistance to protect the embedded secret key and ensure well-behaved operation.

Trusted platforms provide the best properties of open and closed platforms. As with an open platform, trusted platforms allow applications from many different sources to run on the same platform. As with a closed platform, remote parties can determine what software is running on a platform and thus determine whether to expect the platform to be well behaved. The process of dynamically establishing that a platform conforms to the specification expected by a remote party is done through a process called attestation.

Attestation consists of several steps of cryptographic authentication by which the specification for each layer of the platform is checked from the hardware up to the operating system and application code. At a high level, the steps in a basic model of attestation are as follows. A more detailed example is given in Section 4:

- A hardware platform has a signing key K_{sign}. It also has a public key certificate (C_{hw}) for this key.
- When an application A is started it first generates a public/private key pair PK_A/SK_A. Next, the application requests the platform to certify its public key PK_A. The platform uses its signing key K_{sign} to generate a certificate for PK_A. We denote this certificate by C_A. Along with standard certificate fields, the certificate C_A contains the *hash of the executable image of the application A*. This hash is at the heart of the attestation process. The signed certificate C_A is returned to the application.
- When the application A wants to attest its validity to a remote server it sends the certificate chain (C_{hw}, C_A) to the remote server. The server checks two things:
  - The signatures on both certificates are valid and C_{hw} is not revoked, and
  - The application hash embedded in C_A is on the server's list of applications it trusts.

  At this point the server is assured that C_A comes from an application it trusts. The application can now authenticate itself by proving knowledge of SK_A. For example, the application and the remote server could run an authenticated key exchange to generate a shared session key. All communication between the remote server and the application will be protected using this key.

We emphasize that *attestation must result in a shared secret between the application and remote party*, otherwise the platform is vulnerable to session hijacking—an attacker could wait for attestation to complete, reboot the machine into untrusted mode, and masquerade as an authorized application.

Leveraging attestation requires the presence of software that allows the remote party to meaningfully interpret the state of the system. This takes place through a multi-step process whereby the hardware will attest to what operating system it booted, the operating system will in turn attest what application it requires a key for, and will only allow the use of that key by that given application.

Limitations of attestation. It is important to realize that software attestation only tells a remote party exactly what executable code was launched on a platform and establishes a session key for future interaction with that software component on the platform. This does not provide trustworthiness in the usual sense:

- The software component could be buggy and produce incorrect results. The onus is on the remote party to choose who to trust.
- Attestation provides no information about the current state of the running system. For example, attestation does not show whether the software component has been compromised by a buffer overflow attack, infected by a virus, etc.
- Future behavior can only be ensured for authenticated interactions via a shared secret.
- A platform is only as trusted as the tamper resistance of hardware and level of assurance of its trusted OS.

3 An OS for Trusted Platforms

The vision of trusted computing falls apart when it encounters the realities of modern general-purpose operating systems. OSes such as Microsoft Windows and Linux are large and complex code bases optimized over the years for ease of use, performance, and reliability. As a result they are incompatible both in design and implementation with the objective of providing a high assurance platform. High assurance is essential as a trusted OS must instill confidence in remote parties that it can be relied upon to execute their code in a well-specified fashion.

The protection model provided by contemporary operating systems is poorly matched to the needs of trusted computing. In a trusted platform the primary security objective is to isolate subjects from one another. The fine-grained resource abstractions for controlled sharing provided by typical OSes would add needless complexity to a trusted OS, thus detracting from its primary goal of providing secure isolation.

The approach we advocate and have begun to explore in our own work on building a trusted operating system, is to use a virtual machine monitor [14] (VMM). A virtual machine monitor is a thin system software layer that exports the abstractions of virtual machines (VMs) that look like the real hardware.

The simplicity of the VMMs interface and implementation provides the means for building a high-assurance OS that offers strong isolation [17]. VMM's also provide backwards compatibility, allowing existing services and operating systems to realize the benefits provided by trusted platforms with little or no modification. Users can continue to use their normal operating systems for applications that do not require trust from a remote party. Developers building services that require trust can utilize the wide range of existing secure operating systems, applications, etc. , thus allowing them to leverage a huge amount of high quality existing code and development environments.

Our trusted virtual machine monitor (T-VMM) exports two different types of virtual machine abstractions:

Open-box VMs are traditional virtual machines that exactly match the hardware interface of the machine. They are used to run general-purpose operating systems such as Microsoft Windows or Linux and allow the platform owner full access to the hardware state of the VM just as in a normal open platform.

Closed-box VMs provide the same hardware interface as open-box VMs. In addition, a virtual device is provided that allows them to do attestation. To platform owners, the closed-box VMM is a black box. They can grant it access to resources but they cannot inspect or tamper with its contents.

Hardware attestation needs only attest to the fact that the T-VMM is running. For applications to attest, the attestation virtual device can provide a closed-box VM with a signed hash of its executable plus some attributes which it can then present to a remote party to obtain a token encrypted under the public key of the T-VMM. The attestation interface can then be used to decrypt this token, but it will only release the token to the VM whose hash and attributes match those that were originally used to request the token. This token will contain a session key, certificate, or some other means of allowing the VM to authenticate itself.

The T-VMM has total control of both the visibility and use of hardware resources by the VMs. Resource management policy is specified by the platform owner directly to the T-VMM.

Storage devices are abstracted into disjoint virtual disks. Virtual storage can be either encrypted at the block level by the T-VMM or left as plain text in accordance with the performance and security requirements of the VM. Communication devices such as network interface cards can either be virtualized or exported directly to a VM. User interface and display devices are multiplexed among the VMs in such a fashion that one VM cannot observe the user interactions of another.

To support composition of VMs and communicate between VMs, the T-VMM supports the notion of a virtual device. A virtual device can be implemented by a closed box VM and exported as a device to any VM. For example, many closed box VMs will want to export a virtual NIC or virtual serial port to allow other local VMs access to their functionality.

The T-VMM supports a trusted console that allows access to the T-VMM. This is used to control the allocating hardware sources to VMs, mapping of I/O devices to VMs, the destruction of VMs, etc. . The console VM can be accessed via a trusted path. How to securely facilitate this access in a backwards compatible and seamless way is a question we are still are working to address.

4 Example Applications

We survey several areas where trusted platforms promise to have significant impact. We discuss how the introduction of trusted platforms can significantly increase the functionality of existing client side technologies, such as distributed firewalls and massively distributed parallel computing clients. We also look at some entirely novel applications of this technology, like those facilitated by rate limiting. We do not discuss any applications related to Digital Rights Management (DRM) since we find them far less exciting that the applications discussed below.

Regulated Endpoints and Distributed Firewalls. Traditionally firewalls assume that everyone on the "inside" of the network is trusted, while everyone on the outside is untrusted. However, the increased use of wireless access points, tunnels, VPNs, and dial-ins breaks down the distinction between inside and outside. Given today's increasingly dynamic network topologies, distributed firewalls [7] greatly simplify the task of implementing network security policies. With a distributed firewall security policy is defined centrally, but enforced at each individual network endpoint. This supports a richer set of policies and greater scalability than traditional centralized firewalls [15].

On standard hosts, distributed firewalls do an excellent job of protecting a host from others, but are of little use for protecting others from the host—there is no way of ensuring that the host does not simply tamper with or bypass the firewall.

On a trusted platform a distributed firewall is a significantly more powerful primitive since it can prevent packets that violate the central security policy from ever reaching the network in the first place. For example, the distributed firewall can prevent applications that attempt port scanning and IP spoofing from ever reaching the network. Similarly, the firewall can ensure that all VMs on the machine are properly implementing connection rate limits. Hence, distributed firewalls on trusted platforms can provide well-regulated endpoints for a wide variety of different network types.

The architecture for a distributed firewall on a trusted platform is as follows. The distributed firewall runs in its own closed-box VM and listens on a virtual NIC. All packets generated by open-box application VM's on the machine are sent to the distributed firewall VM. The distributed firewall ensures that these packets adhere to the security policy being enforced. If so, it embeds them into an IPsec packet and sends them to their destination on the network. If not, the packets are blocked. The termination point of the IPsec tunnel is the closest network gateway, or alternatively, the remote destination host. The IPsec tunnel is only used to prove to the IPsec endpoint device that the packets are sent via the firewall VM. Consequently, it suffices to use the Authentication Header (AH) in IPsec. There is no need to encrypt the packets.

The main question is how does the IPsec endpoint device know that the sending host is running a distributed firewall. At a high level, the idea is as follows: during initial firewall setup the distributed firewall VM uses attestation to convince a certification authority (CA) that it is an authorized firewall implementing the required security policy. The CA issues a certificate to the firewall VM enabling it to establish IPsec tunnels with peer devices. Without this certificate, peer devices will reject connection requests. Consequently, no application on the machine can communicate with a networked device unless it sends its packets through the firewall VM.

In reality, the exact firewall VM architecture is more complicated. We briefly explain the initial attestation protocol with the CA. We are assuming that the T-VMM on the machine has certified public/private key pairs that can be used for encryption/decryption and for signing. The following steps take place during initial firewall setup:

- the firewall VM generates a public/private key pair PK_{FW}/SK_{FW}.
- The firewall VM requests the T-VMM to sign the hash of the executable image running inside the firewall VM. Let S be the resulting signature. This signature is the main capability used for attestation.
- The firewall VM contacts a CA and sends the public key PK_{FW}, the signature S, and a certified T-VMM public key PK_{VMM}.
- The CA verifies that the firewall executable image (whose signature is S) is an authorized firewall. If so, it issues a certificate $CERT_{FW}$ for the firewall's public key PK_{FW}. The CA also embeds the hash of the firewall executable in the certificate. The CA encrypts the resulting certificate $CERT_{FW}$ under PK_{VMM} and sends the resulting ciphertext $E[CERT_{FW}]$ to the firewall.

This completes the initial firewall setup. Note that no open-box VM can directly use $E[CERT_{FW}]$ since it is encrypted using the T-VMM's public key. Whenever the firewall VM is launched, it first requests the T-VMM's virtual attestation device to decrypt $E[CERT_{FW}]$. The T-VMM does so only if the hash of the executable running in the VM matches the hash inside $CERT_{FW}$. If there is a match, the firewall VM obtains $CERT_{FW}$ which enables it to setup IPsec tunnels with remote hosts. Consequently, when a remote host receives an IPsec session request using $CERT_{FW}$ it is assured that the requesting machine is running an authorized firewall VM.

Rate Limiting for DDOS Prevention. Rate limiting can be used to address the problem of Distributed Denial of Service (DDOS) attacks at both the network and application levels. For example, by limiting the rate at which client machines can issue queries in a P2P network we defend against certain P2P DoS attacks [9]. By limiting the rate at which a machine can open network connection we defend against certain network DDOS attacks [16, 10]. Finally, by limiting the rate at which machines can send email we reduce the rate at which spam email is generated [11].

Implementing a rate limiter with a trusted platform is straightforward. On each trusted platform we run a ticket-granting service in a closed-box VM. The ticket-granting service issues at most one ticket every time quantum. These tickets are content dependent. For example, to limit the rate at which a P2P client issues queries we require an open-box P2P client VM to obtain a ticket from a ticket-granting VM for every query being sent. More precisely, prior to issuing a query, the P2P client VM sends a hash of the query to the ticket-granting VM (via a virtual NIC). The resulting ticket is attached to the outgoing query. The P2P network will discard any incoming queries that contain no ticket or an invalid ticket. Consequently, each client machine can generate at most one query every time quantum (say every 5 seconds).

Without attestation the best known method for achieving these types of rate limits is using client puzzles [11, 4], the practice of forcing a client to perform some costly computation (solving a puzzle) for each request made. A trusted computing solution has several major advantages over client puzzles: no resources must be wasted in order to generate tickets (a real consideration on mobile devices where computing expensive client puzzles could present a significant power drain); users do not need to wait for tickets to be issued; client puzzles vary heavily in their impact based on the type of platform (processor and memory speeds, etc.) whereas trusted-computing based rate limiters are independent of device size or Moore's law.

Improving Robustness via Reputation. Understanding DDOS attacks on today's P2P storage systems requires considering a broad spectrum of attack types. One of the most insidious types of attacks are those based on content poisoning, where a user disseminates damaged or incomplete content (e.g. audio files which have artifacts inserted) in order to make the good content difficult or impossible to find amongst the noise.

One approach to solving these and other problems of mis-behaving users are the use of reputation systems. These are already widely seen in use in online games, P2P file sharing systems [2], and even on eBay to ensure the integrity of sellers. One difficulty with reputation systems is that when users misbehave and their identity is tarnished they can simply apply for a new identity. Without extra infrastructure there is no way to tell whether two distinct identities represent the same entity.

Trusted platforms provide an ideal means of building more robust reputation systems. First, using trusted platforms we can ensure that a single hardware platform represents at most one identity. Consequently, to register multiple identities in a single system one would have to purchase multiple hardware platforms. This approach would thwart common attacks on reputation system where a single platform registers thousands of malicious identities. Second, trusted platforms simplify decentralized reputation systems since the platform can be used to track its own reputation.

Third-Party Computing. Increasingly computing resources are being borrowed, leased, or donated by a third party. Examples of this include (1) using donated cycles for massively parallel scientific and mathematical computations by distributed.net, SETI@home, and Folding@home, (2) using leased time on commercial computer farms for doing large-scale rendering and animation, and (3) the emerging field of grid computing that allows heavy users of scientific computing resources to pool and share their computing resources.

The difficulty with this approach to massively parallel computation is trusting the machines doing the computation to (1) produce the correct results, and (2) keep the contents of the computation secret. Trusted platforms offer an ideal mechanism for solving both problems. Using attestation, remote machines can prove that they are running the expected executable image, the trusted OS will of course keep the computation and its associated state private. The executable can use its token to sign and encrypt the results of its computation, thus ensuring its privacy and authenticity.

Civil Liberties Protection. Increasingly law enforcement requires the use of network surveillance devices [1] that can potentially infringe on civil liberties. Currently, these devices are certified not to exceed their legal boundaries by inviting a select group of experts to review their design. However, there is no guarantee that the system reviewed by the experts is the one deployed in the field. Attestation enables us to do precisely that. Building such devices on a trusted platform enables the platform to prove to third parties that the software on the device is the one authorized to execute. Note that our threat model excludes compromise of the underlying tamper-resistant hardware, which is possibly not beyond the reach of law enforcement agencies.

5 Related Work

The basic mechanisms of attestation have been well studied. Gasser et al. [13] describes an architecture which performs a secure loading process with minimal hardware support to certify to a remote party the operating systems and applications on a platform. Work by Tygar et al. [18] describes host integrity checking with secure coprocessor. More recent work by Arbaugh [6] presents a practical architecture for secure bootstrapping that provides a similar chain of integrity checks to those required for attestation. However, it is important to note that secure bootstrapping and attestation are fundamentally different capabilities. Secure bootstrapping limits what software can be run on a platform, whereas attestation merely reports what software a platform is running.

Prior work has studied attestation in a relatively limited context, usually allowing hosts within an administrative domain to certify what OS they are running to their peers or an administrator [19]. Our much broader vision of trusted computing coincides with efforts such as TCPA [5] and Microsoft's Next Generation Secure Computing Base (NGSCB) project (formerly "Palladium") [8, 3] to deploy trusted computing platforms ubiquitously and provide a very general mechanism for application designers.

TCPA is a platform specification developed by an industry consortium to provide hardware support for trusted computing. Several current implementations of the initial TCPA 1.1b spec have already been implemented in single chips and shipped in the IBM T30 laptops. TCPA does not provide a complete solution to building a trusted platform as it deals strictly with the problem of key management and attestation. Other features required to support a flexible trusted OS (e.g. efficient architecture support for virtualization, additional protection mechanisms, a trusted path to the trusted OS, etc.) are not provided by TCPA. The software model assumed by TCPA is implicitly one of a trusted version of today's commodity OS's. As we have argued this approach is incompatible with the assurance requirements of a trusted platform.

Microsoft's NGSCB project aims to provide hardware [12] and OS support for running authenticated software on an open platform. In NGSCB trusted applications are built on top of a single dedicated trusted operating system specified by Microsoft. This operating system is protected from Windows (but not vice-versa) via special purpose hardware memory protection. All applications are limited to using only this common operating system. In short, NGSCB as it is currently described provides less flexibility and weaker isolation properties than our proposed architecture.

6 Conclusion

We have just begun to explore the broad range of potential benefits that trusted computing can bring to distributed systems. Extending trust from the core of the network to its end-points solves or greatly simplifies many problems in distributed systems as well as enabling a wide range of new applications. In future work we will continue to study the range of systems issues that arise in implementing OS support for trusted computing as well as engage in further study of its applications.

Acknowledgments

The authors would like to thank Ben Pfaff for his editorial assistance and thoughtful comments on this work. This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481 and through support from the Packard Foundation.

References

[1] Fbi. carnivore diagnostic tool. `http://www.fbi.gov/hq/lab/carnivore/carnivore.htm`.

[2] The mojonation p2p platform. `http://www.mojonation.net`.

[3] Microsoft next-generation secure computing base—technical FAQ. `http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/%news/NGSCB.asp`, February 2003.

[4] M. Abadi, M. Burrows, M. Manasse, and E. Wobber. Moderately hard, memory-bound functions. In *NDSS 2003*, february 2003.

[5] Trusted Computing Platform Allaince. Tcpa main specification v. 1.1b. `http://www.trustedcomputing.org/`.

[6] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *In Proceedings 1997 IEEE Symposium on Security*, pages 65–71, May 1997.

[7] Steven M. Bellovin. Distributed firewalls. *;login:*, 24(Security), November 1999.

[8] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft palladium: A business overview. `http://www.microsoft.com/PressPass/features/2002/jul02/0724palladiumwp.asp`, August 2002.

[9] N. Daswani and H. Garcia-Molina. Query-flood dos attacks in gnutella. In *ACM Conference on Computer and Communications Security*, nov 2002.

[10] Drew Dean and Adam Stubblefield. Using client puzzles to protect tls. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[11] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proc. of Crypto 92*, pages 139–147, 1992.

[12] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *Proc. of the 7th Australian Conference on Information Security and Privacy*, pages 346–361, 2002. Springer-Verlag Lecture Notes on Computer Science.

[13] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[14] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.

[15] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[16] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *Proceedings of NDSS 99*, pages 151–165, 1999.

[17] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A retrospective on the VAX VMM security kernel. In *IEEE Transactions on Software Engineering*, November 1991.

[18] J. D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[19] Edward Wobber, Martı́n Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1):3–32, 1994.