Trustworthy Computing and Attestation

Jonathan M. McCune

Collaborators: Yanlin Li, James Newsome, Adrian Perrig, Amit Vasudevan, and others

11.11.11
Is my computer secure?
Externally Verifiable?

- Desirable property: Remotely verify trustworthy device operation

![Diagram showingExternally Verifiable with Yes/No option and sections for Hardware, Operating System, A1, A2, A3]

3
Embedded Systems Example

• Computers are everywhere
  – Uses electricity? Likely to have a CPU.
  – Additional devices are emerging (e.g., thermometers)
• Embedded processors enable new features, BUT
  – Features increase complexity
  – Complexity results in bugs
  – Bugs require software updates to fix (today, anyways)
• Trend: embedded systems become networked
  – Network access enables many features

Scary: Embedded systems with network access and code update features
Example: Vehicular Embedded Networks

• Technology trends
  – Steady increase in number and complexity of processing units
    • Regenerative braking, GPS, in-car entertainment, safety systems
  – Car communication systems
    • DSRC, cellular technologies, BlueTooth, USB, OnStar

• “Ford rolls out software fix for hybrid brakes” CNN 2/4/2010

• Security challenges:
  – Vehicular malware!
Example: Tuning Protection

• Problem
  – Individuals alter engine controller software to get more power from engine

• Consequences
  – Engine damage
    • Who is liable for engine damages?
  – Next-gen vehicle-to-vehicle safety systems
    • Who is liable for crashes?

• Challenge
  – How can we verify the software currently running in the engine controller?
Challenges

• How can we build secure systems with the following properties
  – Highly distributed system
  – Large-scale
  – Networked devices using wireless communication
  – Resource-constrained environment (low-power isn’t just for batteries)
  – Non-expert users!
  – Protects against powerful remote adversary

• This is hard!
Attestation to the Rescue!

- Attestation enables us to verify what software is executing on a potentially untrusted device
- Software code integrity is an extremely powerful property for building secure systems
- Example: Tuning protection using attestation

What SW is running?

\[ \text{Hash(Software)} \]
Generating Attestations

Two basic mechanisms:

• Trusted Hardware
  – Ex: TCG’s Trusted Platform Module (TPM) chip
  – Already included in many platforms (300M+)
  – Cost per chip less than $1

  – AMD SVM: SKINIT instruction
  – Intel TXT/LT: GETSEC[SENTER] instruction

• Software-only approaches
• We will discuss both
TCG Trusted Platform Module (TPM)

- Platform Configuration Registers (PCR)
- Non-Volatile Storage (EK, SRK)
- Random Number Generator
- Secure Hash SHA-1
- Key Generation
- Crypto RSA

DIP Packaging or integrated into SuperIO

LPC bus

I/O
Basic TPM Functions

- PCRs store integrity measurement chain
  \[ \text{PCR}_{\text{new}} = \text{SHA-1}(\text{PCR}_{\text{old}} || \text{SHA-1}(\text{data})) \]

- On-chip storage for Storage Root Key \( K^{-1}_{\text{SRK}} \)

- Manufacturer certificate, e.g., \( \{K_{\text{TPM}} \} K^{-1}_{\text{IBM}} \)

- Remote attestation (PCRs + AIK)
  - Attestation Identity Keys (AIKs) for signing PCRs
  - Attest to value of integrity measurements to remote party

- Sealed storage (PCRs + SRK)
  - Protected storage + unlock state under a particular integrity measurement (data portability concern)
Basic TCG-Style Attestation

BIOS  
Boot Loader  
OS Kernel  
Apps

TPM  
PCR\textsubscript{s}  
K\textsubscript{1}  
Module 1  
Module 2  
conf  
App 1  
App 2

Hardware  
Software
Basic TCG-Style Attestation

What code are you running?

Remote platform

Module 1

Module 2

conf

OS Kernel

Apps

App 1

App 2

TPM

{PCRs}_K_{-1}

PCR~S

Boot Loader

Verifier

K_{-1}
Example: TCG on Linux

- Integrity Measurement Architecture (IMA) by IBM
- Measurement principles
  - Whole system measurements
  - Measure all executable content on-demand
    - Too expensive to measure whole system
    - Content is added dynamically
  - Measure content before execution
    - Only measured content can introduce and measure new content
  - Place as little trust as necessary in measurement system
Part 1: TCB Reduction with Hardware Support
Trusted Computing Base

Attestation and TCB are closely related

• If an attestation does not cover entire TCB, then it only describes partial system state

• Making sense of attestations is hard work (e.g., lots of subtle config variants)

• Goal: trust as little code as possible, and attest only to it
Motivating Example

• Conscientious developer
• Wants to protect critical data
  – Cached account credentials
  – Financial information
  – Mission-critical information
  – Sensor integrity (e.g., camera)
    • Citizen journalism, …
• Evaluates low-cost options
• Her best efforts rest on a house of cards…
Challenge: Reducing the Trusted Computing Base

• Today’s OSes have too much power
• Total access to application data

• App may require little OS support
  – Self-contained computation ‘S’

• Trusted computing base for S includes majority of: OS, drivers, and privileged applications!!!
What is S?

- Self-contained code in an application
- Data secrecy and integrity requirements
- General-purpose computing
- Some examples
  - Manages usernames / passwords for apps
  - Manages Access Control List (ACL)
  - Filters allowable sites when VPN active
  - Platform for introspection into legacy OS
The Flicker System

- Isolate security-sensitive code execution from all other code and devices
- Attest to security-sensitive code and its arguments and nothing else
- Convince a remote party that security-sensitive code was protected
- Add < 250 LoC to the software TCB
Today, TCB for sensitive code S:
- Includes App
- Includes OS
- Includes other Apps
- Includes hardware

With Flicker, S’s TCB:
- Includes Shim
- Includes some hardware
Basic Flicker Architecture

1. Pause current execution environment (legacy OS)
2. Execute security-sensitive code using *late launch*
3. Preserve session-state with TPM sealed storage
4. Resume previous environment

- Not the intended use of late launch, sealed storage
- Intended use is an infrequent, disruptive event
  - Use to replace lowest-level system software
  - All but one CPU must be halted for late launch
- Our use resembles a context switch
  - Setup protected execution environment for sensitive app
  - Late launch and TPM sealed storage on the critical path
Flicker Execution Flow

Flicker Session

Load Flicker driver
Accept uninitialized SLB
Initialize SLB
Suspend OS
SKINIT
Execute PAL
Piece of App. Logic
SLB
Cleanup
Extend PCR
Resume OS
Return outputs

PCRs
TPM
K₁
Attestation

TPM

PCRs:

S
Shim

STOP

0
Inputs
0
Outputs
0

K_{-1}

Time

PCRs

TPM

K_{-1}
Attestation

What code are you running?

PCRs:

Inputs

Outputs

Versus

Sign(, K_{-1})
Context Switch with Sealed Storage

- Seal data under combination of code, inputs, outputs
- Data unavailable to other code
Application: Rootkit Detector

- Administrator can check the integrity of remote hosts
  - E.g., only allow uncompromised laptops to connect to the corporate VPN
Next-Generation TCB Minimization

• Based on a special-purpose hypervisor
Meet TrustVisor

• Originally developed for x86 platforms
• Tiny hypervisor for isolation of code S
  – No scheduling or Inter-Process Communication
• Efficient transitions between OS and S
• External verification of Output = S(Input)
• Protected storage for S
Protected Storage

• Initially, S is “red” (untrusted)
• App can register S \(\rightarrow\) “blue” (attestable)
• TV enables “blue” code to protect data…

![Diagram showing app level security](image)

- Access-controlled by identity of S (hash)
- Enabled by TPM-like Sealed Storage
- “Micro-TPM” in software
## Alternative Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Metric</th>
<th>TCB Size (LoC)</th>
<th>Protection granularity</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic kernel</td>
<td>best</td>
<td>millions</td>
<td>consistent</td>
<td>code</td>
</tr>
<tr>
<td>Virtualization</td>
<td>good</td>
<td>millions</td>
<td>process</td>
<td>consistent</td>
</tr>
<tr>
<td>Virtual TPM (vTPM)</td>
<td>millions</td>
<td>consistent</td>
<td>process</td>
<td>consistent</td>
</tr>
<tr>
<td>Security /μkernel</td>
<td>moderate</td>
<td>~100K</td>
<td>Flicker</td>
<td>fine</td>
</tr>
<tr>
<td>TrustVisor</td>
<td>fine</td>
<td>&lt;1K</td>
<td>TrustVisor</td>
<td>fine</td>
</tr>
</tbody>
</table>

TrustVisor x86 runtime TCB in lines of code:
- ~6500 C/ASM + ~2800 Headers
- Hypervisor + crypto
TrustVisor ↔ OS Architecture

TrustVisor:
- Virtualizes RAM, CPU
- Restricts DMA
- Restricts TPM to Locality 1

- App 1 … App n

OS

Device Drivers

Locality 1

TM Driver

Locality 2

Hardware
TrustVisor ↔ S Architecture

- TrustVisor API
  - Registration
  - Invocation
  - Micro-TPM

DMA Devices (Network, Disk, USB, etc.)

TPM

CPU, RAM Chipset
Identifying S to TrustVisor

• Applications identify S via **registration**
  – Page-level protection granularity
• Applications make “normal” function calls
  – TrustVisor detects switch to S via traps
• S runs with no access to legacy OS
  – One set of Inputs and Outputs per **invocation**
Sensitive Code Timeline

Multiple *invocations* during a single *registration* cycle

Initialize TrustVisor
Application Starts
Register S
Invoke S: SSL Session Init
S Complete: Session active
... 
Invoke S: SSL Session Init
S Complete: Session active
Unregister S
Application Exits

S’s Runtime State Protected
Micro-TPM Design

• Small subset of hardware TPM operations for:
  – Protected Storage + External Verification

• TrustVisor implements TPM-like operations in software on main CPU
  – Extend, Seal, Unseal, Quote, GetRand

• Trust in Micro-TPM requires root of trust
  – Hardware TPM on x86 platforms
  – Multiple options on mobile platforms, e.g.,
    • TCG-specified Mobile Trusted Monitor (MTM)
    • Software-based attestation
    • Signed code
Ongoing Work

• On top of isolated execution capabilities…
  – Application-specific modules to perform sensitive operations (e.g., key management)
  – Software TPM for remote attestation
    • Contemplate use of TCG-specified MTM
  – Binding data to particular code identity
  – Trusted path for I/O to human user
    • Certain peripherals dedicated to application-specific modules under certain conditions (e.g., approve txn)

• Continue to look for inexpensive platforms
  – Especially interested in bringing similar security properties to mobile platforms
TrustVisor Conclusions

• Tiny hypervisor to support isolation
• Hardware support can strengthen properties
  – We plan to become experts on capabilities and availability of such support
• TPM-style operations in software
• Compelling performance argument
• Require minimal OS changes
  – Today’s servers are tomorrow’s smart phones
  – HW virtualization support likely to trickle down
• Foundation for future trustworthy systems
Part 2:
VIPER: Verifying the Integrity of PERipherals’ Firmware
Motivation

• Triulzi injected Malware into a Tigon NIC to eavesdrop on traffic (2008)
• Malware on NIC deploys malicious code into GPU, causing GPU to store and analyze data sent through NIC
Motivation

• Chen injected key logger into Apple Aluminum keyboard firmware (2009)

• Buffer overflow vulnerability in Broadcom NIC was disclosed (2010)
Malware on Peripherals

- Eavesdrops on data handled by peripherals
- Modifies executable programs or scans data in main memory through DMA if IOMMU is not perfectly configured
- Spread malware to other peripherals through DMA
- Collaboration with malware on other peripherals through communication through PCI bus
Challenge & Problem Definition

• Open challenge to detect malware on peripherals
  – Limited memory and computational resources on peripherals
  – Hardware-based protection is expensive and impractical

Verifying the integrity of peripherals’ firmware, and guaranteeing absence of malware
Contributions

1. Systematically analyze malware features on computer peripherals
2. Propose VIPER, a software-only primitive to verify integrity of peripheral devices’ firmware
3. Propose a novel attestation protocol that prevents all known software-only attacks
4. Fully implement VIPER on a Netgear GA620 network adapter on an off-the-shelf computer
Assumptions & Attacker Model

- **Assumptions**
  - Physical attacks are out of scope
  - Verifier Program on host CPU is protected & trusted
  - Verifier program knows peripherals’ information

- **Attacker Model**
  - Compromises peripherals’ firmware
  - Controls remote machines to assist the compromised device
  - Cannot break cryptographic primitives
Software-based Root of Trust

- Verifier verifies checksum & timing results
  - Malicious code or operations either result in invalid checksum or require longer computation

Diagram:

Host CPU
- Verifier Code
- Checksum Simulator
- Expected Firmware
- Timer

Peripheral
- Verification Code
  - Checksum Function
  - Communication Func
  - Hash Func

1. nonce
2. Untampered environment and Compute Checksum
3. checksum
4. hash
Proxy Attack

• Proxy Helper: a remote machine
  – Has a copy of correct firmware
  – Computes expected checksum for untrusted device
VIPER: Challenges

• Local Proxy Attack
  – Peer-to-peer communication between two peripherals through DMA
  – A faster peripheral helps a slower peripheral

  **Verify faster peripheral first!**

• Remote Proxy Attack
  – E.g., a NIC can communicate with a remote proxy helper over Ethernet

  **How to defend against a Remote Proxy Attack?**
Software-based Root of Trust

• Verifier verifies checksum & timing results
  – Malicious code or operations either result in invalid checksum or require longer computation
Latency-Based Attestation Protocol

Normal Case:

Host CPU

Peripheral

Proxy Attack:

Host CPU

Peripheral

Proxy Helper

Time line

Overhead

$T_{send}$

$T_{recv}

$T_{comp}$

$T_{helper}$
Can we defend against a proxy attack all the time?

- **Parameters**
  - Computation time on proxy helper: \( T_{\text{comp}}^{\text{proxy}} = \text{zero} \)
  - Communication time of a proxy attack: \( T^{\text{proxy communication}} \)
  - Checksum computation time: \( T^{\text{peripheral checksum}} \)
  - Timing accuracy on host CPU: \( T^{\text{cpu accuracy}} \)

\[
T^{\text{proxy communication}} > T^{\text{peripheral checksum}}
\]

\[
T^{\text{proxy overhead}} = T^{\text{proxy communication}} - T^{\text{peripheral checksum}}
\]

\[
T^{\text{proxy overhead}} > T^{\text{cpu accuracy}}
\]
Parallel Computation & Transmission

• Host CPU sends next nonce before the peripheral returns checksum
• The new nonce determines which checksum to return
  – Proxy helper cannot know which checksum to return, so has to return all checksum states that have been updated
  – Increases overhead of a proxy attack
VIPER

- Latency-based attestation protocol
  - Multiple nonce-response pairs
- From faster peripheral to slower peripheral
Implementation

• PCI-X Netgear GA620 NIC
  – Two MIPS Microcontrollers (200 MHz)
  – 4 MB SRAM
  – Open Firmware Version 12.4.3
  – Checksum and communication code: 656 MIPS instructions
  – SHA-1 Hash Function: 2 KB binary

• Sun Fire rack-mount server
  – Single-core AMD Opteron Processor
  – 2 GB RAM, Two PCI-X slots
  – Linux 2.4.18
Verification Procedure

1. Verify entire scratch pad memory
   - PC stays within the trusted code

2. Verify checksum and hash func
   - CPU A and CPU B cannot access each other’s scratch-pad memory

3. Compute hash over Firmware Contents
   - No hash func
   - CPU B
   - Only verify Scratchpad memory

1. CPU A and CPU B cannot access each other’s scratch-pad memory
2. Attestation can start from either A or B
Evaluation Results

Threshold (4.5% over benign case)

Benign Case

Various Attacks
VIPER Conclusions

- Detecting malware on peripherals’ firmware becomes increasingly important.
- Extend previous software-based root of trust mechanisms to defend against proxy attacks.
- Implementation & evaluation on a Netgear GA620 NIC.
- Anticipate that these techniques will make software-based root of trust practical on current platform.
Q & A

• Thank you!

• jonmccune@cmu.edu

• Papers all available online: http://www.ece.cmu.edu/~jmmcccune