System Security

76432 characters in 11785 words on 1885 lines

Florian Moser

August 23, 2021

1 Introduction

real systems are more complicated than alice, bob and eve
multi-tenancy, computation outsourcing, user interfaces, ...
security aspects in hardware/software, design/implementation, digital/physical world

1.1 terminology

integrity
data not changed by unauthorized party
either prevent or detect modification

confidentiality
unauthorized party does not understand data
data looks like random bits
secrecy (data belongs to sender only; own data)
confidentiality (data belongs to some specific users; customer data)

availability
data is accessible by parties
distribution of service (DoS) by exploiting bugs
or overwhelming service with too many requests

authentication
identity of sender is verified

authorization
requester has capability (= is entitled) to use service/data
precondition is valid authentication

1.2 cryptographic primitives

crypographic hash function
one-way (given y , find x' such that y = H(x'))
weak collision resistance (given x, cannot find x' such that H(x) = H(x'))
strong collision resistance (cannot find any x, x' such that H(x) = H(x'))

1.3 symmetric crypto

secure key length around 128bit

for confidentiality
→ encryption key
stream cipher processes message bit by bit
block cipher processes message in blocks (like AES, DES)

for authentication
→ authentication key
compute message authentication code MAC c = C(K, m)
receiver accepts m if same c recomputed with shared key K

1.4 asymmetric crypto

secure key length around 3072 bits for RSA, 256 bits ECC

vs symmetric crypto
key distribution easier (can share public keys freely)
only way to authenticate source of data
much longer key sizes
much slower (around 100 for ECC - 1000 times slower)

for confidentiality
→ public key encryption
message encrypted with public key c = E(K_{public}, m)
receiver decrypts with private key m = D($K_{private}$, c)

for authentication
→ digital signature
message with appended signature s = E($K_{private}$, m)
receiver accepts if m = D(K_{public}, s)

1.5 reverse engineering

black box tools
blobs with strings
running processes with ps
system calls with strace
network traffic with wireshark
open network connections with netstats
open files / ports with lsof
but no internals, network-level obfuscation

static analysis
cutter analyses control flow
ghidra additionally tries to decompile to c
but need manual work against source code obfuscation

debugging
investigate used data & taken control flow branches
but only one trace, can easily get lost

possible approach
use black box tools to get general overview
use debugging to identify commands / command parsing
use static analysis to understand critical parts

1.6 general attacks

credential sniffing
find out passwords
especially useful when same combination used in multiple system
collections of used passwords like exploit.it, haveibeenpwned.com, rockyou.txt

replay attacks
cryptographic keys allow encryption / authentication
but do not guarantee integrity / freshness
need to prevent replay of messages (and reexecution of commands)

capture the flag (CTF) challenges
test skills in binary exploitation / other exploits
good tutorials by adam doupe

2 security protocols

2.1 attacker models

capabilities of attacker are clearly formulated
then shown property of system holds under that attacker
like "attacker + hardness assumption ⇒ protocol + property"

potential property
indistinguishability (attacker decides b = {0,1} of c = enc(m_b))
key recovery (attacker outputs key used in enc/dec)

oracles
ciphertext only (COA); some c kown
known plaintext (KPA); some (m, c) pairs known

interactive oracles
when attacker can ask some oracle to perform operation
chosen plaintext (CPA); can ask for c = enc(m) of some m
chosen ciphertext (CCA); can ask for m = dec(c) of some c
chosen cipher- and chosen plaintext can do both

2.2 encrypt and compress

encrypt then compress
output of encryption should be randomized
hence on average, no compression possible

compress then encrypt
might leak data redundancy (by length of ciphertext)
like CRIME attack

2.3 encrypt and authenticate

for MAC authentication, E encryption
encrypting message m, to receipt r

authenticate-then-encrypt
r = E(MAC(m) || m)
gives secrecy & authenticity
but vulnerable to CCA by observing system
like "padding oracle" as shown in TLStimmig
but vulnerable to DoS (invalid messages detected only after decryption)

encrypt-then-authenticate
c = E(m), r = c || MAC(c)
gives secrecy & authenticity, provably secure
can immediately detect crafted message with MAC

encrypt-and-authenticate
c = E(m), r = c || MAC(m)
same attacks as authenticate-then-encrypt
additionally leaks cleartext (as MAC not required to hide it)
like MAC' = m || MAC (for MAC provably secure)
used in practice (like SSH); not necessarily insecure

2.4 iterate over locking system

lock opened by secret of key

considerations
encrypt/authenticate secret
but attacker can still replay
include freshness into secret
but needs acceptable window (like 10 seconds..?)
instead of key-generated freshness, use challenge of lock
but attacker can relay (MitM)

distance bounding
measure roundtrip time for challenge response
reduce variance of trip time enough to be precise

protocol proposal
key sends open/close signal to lock
lock generates nonce and sends it to key
key encrypts nonce with shared key and sends cipher to lock
lock decrypts and checks plaintext is nonce, then executes request
but always same shared key

single key protocol proposal
with each signal, lock generated new secret
secret encrypted under old secret and sent to key
key uses this new secret for future commands
but impossible with multiple keys

3 systems

3.1 vocabulary

instruction set architecture (ISA)
specification of software / hardware interface
includes register / main memory size
like x86, ARM, RISC-V

microarchitecture
implements ISA
defines caches, branch prediction, reorder buffer, ...
like intel core i7, AMD Ryzen, ...
no direct access to details, but might leak information

3.2 performance

memory wall
access to memory is too slow
requirement to hide memory latency motivates many optimizations
like caches, pipelines, branch prediction, out-of-order execution

3.3 power consumption

transistor
connection between pull up (pMOS) and pull down (nMOS) networks
if open then networks connected (power used)
else no power used (besides some leakage)

dynamic power consumption $P_{dynamic}$
for voltage V_{dd} and transistor capacity C
power to charge once is $C * {V_{dd}}^2$
if at frequency f all switch from 0 to 1 (and back)
then need $P_{dynamic} = 0.5 * C * {V_{dd}}^2 * f$ energy

static power consumption P_{static}
power consumption when no gates are switching
caused by quiescent supply current I_{dd} "leakage current"
then need $p_{static} = I_{dd} * V_{dd}$ energy

CMOS gate
basic building block out of which logic gates are constructed
needs two transistors C1 and C2
if value does not change (0→0 or 1→1)
then no transistor changed, hence no power used
else power consumed relative to C1 or C2

logic gates
built out of transistors
like single-input (NOT, buffer)
like two-input (AND, OR, NAND, NOR, XNOR)

instruction power consumption
storing data (depending on #1 in operand)
shifts and rotations (depending on size of operand)
logical / arithmetic operations (depending on values)

3.4 direct memory access (DMA)

CPU can grant DMA permissions to devices
after granting permissions, memory accesses unchecked
improves RAM access speeds, but cannot prevent invalid accesses
hence can dump memory and extract sensitive information

firewire
high-speed serial bus, useful for real-time applications
uses DMA if driver supports it (which is usually the case)
if not IOMMU, need to destroy / disable port & protocol
incl. others using the same protocol (like thunderbolt)

IOMMU
setup by OS, introduced to control DMA access
maps device addresses to physical address
constrains to only access valid DMA targets

3.5 x86 system

used in servers, computers, laptops, ...

instructions
op dest src (intel syntax)
dest/src could be register, memory location or constant
add eax ebx (add & store in eax)
mov eax, [ebx] (move from location ebx into eax)

platform overview
processor (with one to many CPUs called cores)
chipset which connects processor to memory (RAM) & peripherals
peripherals using various bus-interfaces
like CPU connects cores, DDR, display ports
like chipset connects VGA, PCIe, SATA, USB, ETH, ...

core components
memory management unit (MMU)
programmable interrupt controller
cache for efficient memory access
virtual machine extensions (VMX)
connection to other cores & chipset

current privilege level (CPL)
CPU tracks CPL using 2 register bits
ring 0 for kernel, ring 1 & 2 for drivers, ring 3 for applications
currently drivers part of kernel, hence only ring 0 & 3 in use
used to limit access to certain instructions, IO
used to protect kernel memory (legacy)

page tables
to preserve integrity / confidentiality must not share memory
use page tables to assign physical memory to applications / kernel
applications work with virtual addresses, translated to physical by MMU
kernel configures page tables (map virtual to physical memory)
kernel can access own pages, applications can access their own
supervisor bit determines if ring 0 or others able to access
RW bits differentiate between read / write page
execution disable (ED) bit determines if page can be executed

cache
ensure repeated access to same data is fast
big performance difference in cache hit vs miss

organized in levels (L1, L2, L3)
with increasing level size goes up, speed goes down
single L1, L2 per core, L3 shared
shared across all applications & kernel
cache location depends on data address
new content replaces old if cache already full

pipelining
split instructions into smaller steps
like fetch (IF), decode (ID), execute (EX), memory access (MEM), write back (WB)
can run these in parallel with other instructions
current CPUs have > 20 pipelines

out of order (OoO) execution
parallelize execution stage to fully utilize all execution units
reorder buffer resolves dependencies & schedules instructions
need to retire (but not start!) in-order
will check exceptions only during retire

branch predictions
static predictions known at compile time
dynamic predictions based on runtime / last time branch behaviour
use branch target buffer (BTB) to store runtime data
not flushed on context switch

virtual memory
each process has illusion of having all system memory
actual physical memory is shared between processes
MMU translates virtual pages to physical addresses
using a hierarchy of page tables (managed by kernel)
but walking these page tables is expensive
page table entry (PTE) contains permission bits (execute, read-only)

memory access
check address is cached
if not, walk page tables
then request physical address content
save & cache value + cache page table walk
finally check if permissions OK
if yes return, else raise exception

3.6 ARM

ARM Ltd develops & licenses architecture
manufacturers incorporate design into products
cheaper, less power usage than x86
commonly used in smartphones, some also in servers

history
1980 british manufacturer, first as co-processor of CPU
1990 design team spin off to ARM Ltd for smartphones
2000 intel failed to compete in mobile market
2016 acquired by SoftBank
2022 nvidia will buy ARM

evolution
operator requirements (subsidy locks, copy protection)
regulator requirements (RF type approval, theft deterrence)
need immutable ID, device authentication, secure storage, ...
manufacturers forced to implement security measures for compliance
2001 J2ME, 2002 ARM TrustZone, 2005 Symbian platform security, ...

System on a Chip (SoC)
includes CPU, 4G model, WiFi, Bluetooth, memory, ...
bus connects CPU with on chip-memory, memory controllers
and more devices (like 4G, ...) outside the SoC

3.7 platform security

sudo CVE
sudo sets user id of executing process
sudo itself runs as root
but syscall did not change value with -1 userid
then sudo simply executed as root

intel management engine
obfuscated binary running directly on CPU (OS independent)
with own TCP/IP stack
allows to login, turn on/off, monitor
priviledge escalation CVE allowed arbitrary code execution
zero-touch provisioning updates firmware w/o certificate check
unclear if (more) backdoors are built in

3.8 unix access control

file/directory assigned to owner and group with permissions
permissions can only be changed by owner
root able to do anything, regardless of ownership/permissions

octal notation
set with chmod, like chmod 0600 file.txt
first char for stick bit (1), setguid (2), setuid (4)
2nd, 3rd, 4th for execute (1), write (2), read (4)

symbolic notation
retrieved with ls -l
letter for filetype (- regular file, c character file, d directory)
3 letter group for each owner, group and others
each group read (- or r), write (- or w) and execute (- or x)

evaluation
if owner matches, then owner permissions evaluated
if group matches, then group permissions evaluated
else other permissions evaluated

examples
for directories, read with ls, write with touch, execute with cd
for files, read with cat, write with touch, execute with ./

"sticky bit" on directory
with write access on directory, can rename/delete any file within
hence can circumvent missing file write access
use "sticky bit" so only owner of directory / files themselves can
last char of symbolic notation changes to t or T (executable yes or no)

"setuid"/"setguid" bit on executables
declares that the executable is executed as the owner/group of the file
disabled for scripts, protects processes from modifications
last char of symbolic notation changes to s or S (executable yes or no)
like ping has setuid as root (but non-root can invoke ping)

sudo / su
sudo gives root access to user (if in sudoers file)
need to enter own password (can be turned off in config)
su allows to impersonate other user (like su bob)
need to enter password of impersonated user

example password change
/etc/shadow contains hashes of passwords; only root can read/write
/usr/bin/passwd onwed by root & setuid set; all can read/execute
normal user calls passwd executable which then edits shadow file

4 side channels

cryptosystem analysis observes input → crypto operation → output
but one can also observe the device executing crypto operation
like power, time, heat, sound, electromagnetic radiation ...

4.1 attacks

in general possible when resources are shared
between different security domains (CPU, cache, memory)
also applies more generally (air, sound, vibrations, ...)

attack distance
maximum distance under which side channel attack is possible
but might be able to increase it with prof. equipment

vulnerable devices
the simpler the device, the more vulnerable it is
as easier to isolate components under attack
like smart cards

defense
minimize dependence of execution on input (static execution time, ...)
introduce noise (but hard to get right; statistical filtering)

4.2 analysis types

simple side channel analysis
side channel output depends only on key
sometimes trivial, sometimes needs statistics

differential side channel analysis
side channel output depends on key and additional input
usually needs statistics to get to key

4.3 RSA timing

assume system simulatable to get timing reference

assume victim signs attacker-chosen m (by CPA property)
hence so-called signature oracle available

square-multiply
does x = x^2 for each bit, multiply it to result if key bit set
if key bit 0 then runtime lower, else higher
hence finding hamming weight easy
but not much help as 0/1 bit count will be similar

modular multiplication
faster than classic way (tmp = x * m, x = tmp mod N)
as reduction only on demand (if intermediate too big)
hence conditional on x*m

finding exact key with montgomery
assume d0, ..., $d_{(i-1)}$ are known
simulate execution up until d_i
check if montgomery reduction at step i
assuming $d_i == 1$, if reduction needed m into M_1, else M_2
assuming $d_i == 0$, if reduction needed m into M_3, else M_4
measure $diff_1$ = Mean(M_1) - Mean(M_2)
measure $diff_2$ = Mean(M_3) - Mean(M_4)
if $diff_1 > diff_2$ then $d_i == 1$, else $d_i == 0$
bc diff is bigger where bit i predicted correctly

defend against montgomery attack
choose random X per message
SIGN(m) = $(m * X)^d * (X^{-1})^d$ mod n
hence attacker can no longer determine signed number
for performance, compute $(X^{-1})^d$ in advance
still two additional multiplications needed

4.4 power analysis types

measure power consumption repeatedly during execution
need modified reader to provide input
need oscilloscope to measure power consumption
useful for smartcards, RFID, sensor nodes

general approach
in general not able to differentiate individual transistors
but can observe patterns (like difference of square and multiply)
leads to many measurements which are correlated
more than other side channels (like execution time)

simple power analysis (SPA)
evaluation of single execution trace
when key directly determines instructions (hence power consumption)
like square-multiply algorithm, where multiply only follows if input bit 0
then can differentiate key bit 0 (one peak), and key bit 1 (two peaks)
defend by executing always the same instructions

differential power analysis (DPA)
statistical analysis of multiple measurements of crafted messages
when key together with input plain-/ciphertext determine instructions
like stores (#1 determine power usage)
when instruction power consumption depends on value of operands
like shifts & rotations (depending on size of shift)
like logical

high-order DPA
complex statistical analysis of multiple measurements

4.5 power analysis attacks

RSA
power usage differs if squaring or multiplication required
mongomery square (bit 0) vs square+mult (bit 1)
hence by eye expect one peak for bit 0, two for bit 1
find out whole key bit-by-bit

hamming weight of key due to load
used HC05-based smartcard and measure power trace
could determine hamming weight of key of smartcard
as hamming weight = power consumption of #0→1 switches
not that dangerous (bc keys should have high entropy)
but dangerous for plaintexts (as low entropy enables guesses)

advanced DPA attack on DES
DES uses multi-round block cipher to encrypt data
in each round new key used; generated from encryption key
key generation process reads key / rotates it in every round
but can still setup set of equations and read out key
complex statistical analysis of multiple measurements

4.6 power analysis defense

reduce correlation
operand value / power consumption should not correlate
but cost / benefit hard to determine

desynchronization
inject random dummy instructions
but can be removed using SPA and neutralized from waveforms

noise generator
add generator which inserts random noise
but can be filtered out with more measurements

physical shielding
power input detects malicious acts
but false positives

software balancing
insert instructions in low-cost paths
but significant speed reductions

hardware balancing
ensure all instructions have same power cost
but prohibitively costly, hard to design

shamir's countermeasure
decouple power consumption from charging (like internal power source)
two capacitors C1, C2 serve as up network one after the other
gates which change if C connected to power supply or controller
while one recharges, the other powers the microcontroller

4.7 RSA acoustic

different secret keys cause different sounds

setup
laptop as target
microphone close, or 4m with professional equipment
ability to provide chosen ciphertext (victim decrypts)

interesting sounds
high-frequency sounds produced by voltage regulation circuit
caused by vibrations of electronic components
proxy for power consumption

microphones
operate with kHz, but CPUs with Ghz
hence need to find longer patterns (such as modular exponentiation)
indeed can measure different frequencies for MUL, ADD, HLT, ...
works with different PCs / standard microphones
some calibration has to be done

GnuPG attack
implementation calculates mod p, then mod q, then uses CRT
microphone can differentiate when mod p and when mod q
each p / q has different frequency patterns
can differentiate if attacked bit is 0 or 1
extracting 2048 bit key takes ~ roughly an hour

conclusion
GnuPG already has side channel mitigation techniques & constant time
both not enough; need masking techniques

4.8 electro-magnetic pulses (tempest)

electromagnetic emanations (em) used to detect secrets
generated by everthing (like keyboards, cables, processors)

examples
computer screen (demonstrated through plasterboard walls)
wired / wireless keyboard (all vulnerable)
reflections from spoon, human eye, softdrinks, ...
faraday cages would help, but block all wireless traffic

4.9 cache-misses

if victim shares same cache (like js, shared hosting, ...)
cache miss leaks information

flush + reload (shared memory)
attacker flushes memory region
starts victim & waits for completion
accesses target value
if fast, then victim accessed value, else not

prime + probe (no shared memory)
attacker fills memory region with own data

starts victim & waits for completion
access own data again
wherever slow, victim accessed region

on AES
s-boxes are lookup tables used in each round of RSA
byte in key XOR byte in plaintext gives S-box index
measure time until plaintext byte found which takes longest
then on local machine, find key byte that takes longest
possible byte by byte, hence complexity O(k) (down from $O(2^k)$)

4.10 speculative execution (meltdown)

execute instructions on value from invalid fetch
use cache side channels to reconstruct value
possible as value is computed upon before page table walk finished
hence exception only raised after micro ops have been executed

setup
mov eax, $[kernel_{address}]$
(will prefetch & raise exception late)
mov ebx, $[probe_{array} + 4096 * \text{eax}]$
(will fetch cache line of in $probe_{array}$)

exploitation
check which $probe_{array}$ value is accessible fast
then this is the location determined by eax
hence can learn value of $kernel_{address}$

attack vector
"microarchitectural attack"
while architectural state is consistent
state of microarchitecture is not rolled back after exception

4.11 branch predictor (spectre)

branch predictor executes ops despite branch will not be taken
targeted prediction manipulation from different process possible
as content not flushed on context switch

variant 1 (attacker code injection)
attacker-controlled condition (like (x < x2))
with protected access on true (like $probe_{array[kernel[x]*4096]}$)
train branch predictor for true (by choosing valid x, high x2)
then evict x2 from cache (so speculative execution is started)
and choose x to get interesting offset (but false condition)
speculative will access (cache side channel successful)
but not raise exception (as branch not "really" taken)

variant 2 (no code injection)
recreate same branch source / target pattern in attacker process
take branch often to train BTB
evict victim code cache to enter speculative execution
watch BTB take branch as trained for

5 tamper resilience

protect selected critical functionality
like generating/using keys/signatures

5.1 classification

tamper resistant ("bank vault")
to prevent break-in
make attacks slow/expensive
use special/hard materials
like smart card, ATM

tamper responding ("burglar alarm")
to detect intrusion real-time & immediate response
sound alarms and/or erase secret data
applicable to small devices as no heavy hardware required
but devices need battery / communication / erase data
like cryptoprocessors

tamper evident ("seal")
to detect intrusion
after break-in evidence of such is left behind
use chemical/mechanical means
like cryptoprocessors, seals

5.2 FIPS 140-2

protect plaintext keys & critical security parameters (CSP)

level 1 (software only)
security requirements (like specific algorithm / security function)
no physical security beyond basic production-grade components
like personal computer encryption board

level 2 (+tamper-evident)
require tamper-evidence before physically accessing CSP
like pick-resistant locks / doors & seals

level 3 (+tamper-resistance & responding)
physical access to CSP need to be prevented & responded to
appropriate response (like 0-ing all upon detection)

level 4 (+roboustness)
access prevention & response with very high probability
appropriate response (like 0-ing all upon detection)
must work within uncontrolled physical environment

5.3 smart cards

holds secret keys
access protected with PIN
can perform some crypto functions (key use/generation)
protected against side channels

applications
place piece of trusted hardware / secret key at user
authentication (but not decryption of keys) might use a PIN
like GSM (SIM card), ATM (banking card)

limitations
not that efficient to combat fraud
hence need to combine with surveillance, transaction logs, blacklisting
secret keys not encrypted (PIN only unlocks)
hence user must not be able to extract

photonic emission side channel
allows to capture crypto keys
detect accessed AES S-Box to reveal key

5.4 hardware security module (HSM, crypto co-processor)

holds secret keys
access protection with (master) key
can perform many crypto functions, TPM functionality
own power / clock to protect against side channels
active protections against tampering / side channels

applications
so far limited
used in banks to verify keys securely
might change with cryptocurrencies, digital assets, ...
like IBM 4758

interaction policies
ensure encryption / decryption not abused
challenging because signed message is binary blob
"out of context" might sign/decrypt something not intending to

security API attacks
even if hardware secure, security API might be flawed
security API wraps crypto API while enforcing policies
but unsound access policies, API leaking secrets or broken primitives
need cryptoanalysis (flaws in primitives)
need protocol analysis (flaws in protocol / user-exposed API)

5.5 HSM PIN attack

assume network attacker in a bank
can sniff encrypted PIN & plain account number
target to get HSM to reveal PIN with API queries

PIN generation
happens within HSM
account number (PAN) encrypted using bank key
take first HEX values of encrypted PAN
HEX to decimals by decimalization table DT (like A→0, B→1,...)
decimalized PIN + PIN offset = user PIN
(PIN offset useful so user can change PIN)

PIN verification
as input need encrypted PIN, PAN, DT, PIN offset
use PAN, encrypt, apply DT → PIN
use encrypted PIN, decrypt, add PIN offset → PIN'
accept if PIN & PIN' match

decimalization attacker

extract decimalized PIN by passing malicious DT / PIN offset
(1) change entry in decimalization table
if PIN still valid, entry unused, else used and goto (2)
(2) add PIN offset to revert DT change, try out all positions
when PIN is valid again, found out place of value

decimalization example
start with DT = 0123..., PIN offset = 0000 (for simplicity)
change DT from 0123... to 1123...
observe that PIN now invalid
try PIN offsets 1000, 0100, 0010, 0001
until PIN is valid to reveal position of 0 in decimalized PIN

5.6 HSM ISO-0 attack

attack on transformation function of different PIN block formats
input is encrypted PIN block, PAN, in & out key identifiers
output is re-encrypted PIN block in new format

PIN block format
defines how PIN is actually stored
different formats available → transformation functions exist

translation function to ISO
input encrypted PIN block (EPB), PAN
PB = decrypt (EPB), then PB XOR PAN = 04PPPPFFFFF for PPPP actual PIN
if PPPP is not valid PIN, terminates with error
else continues processing

abuse error message
modify PAN with x at some position 00x00000 like 00500000
for x too high, XOR produces non-digit → error message raised
hence can check (P XOR x) < 10
then bruteforce possible values of x

attack analysis
derive PIN within expected 13.6 steps
hence very fast, computationally simple
but need physical access to device / network

prevent attack
access control (limit functionality to what is strictly required)
formally verify security API to prevent further leaks

5.7 crypto tokens

smartcards / USB dongles which hold/protect keys

fixed operations
specify only key ID & additional input
encrypt, decrypt, export

API abuse
export(K1, use K2) → C
decrypt(C, use K2) → get K1
⇒ token does not know C contains K1

PIN unchecked
when in possession of chip card, do not need PIN card
as subprotocol deciding on authentication method is unauthenticated

6 security of commodity systems (PC)

protect security-sensitive applications on commodity systems
with small as possible trusted computing base (TCB)

6.1 security properties

composed out of data, volatile & persistent data
require hardware resources (CPU, memory, peripherals)

launch-time integrity
ensure pristine application started
need integrity (like hash) of initial code/data
to protect volatile data & code
like secure boot

run-time isolation
ensure no interference from malicious OS/applications
need prevention of unauthorized modification of code/data
need prevention of run-time attacks which modify control flow
to protect volatile data & code
like page-based security (r/w/x bits, assigned to ring)

secure storage
ensure persistent storage is not tampered with
need confidentiality & integrity protection
to protect persistent data
like disk encryption

implementation challenges
where to implement functionality (OS, hypervisor, CPU, ...)
how to protect security functions themselves

6.2 trusted OS based solution

peripherals & applications untrusted
but assume OS & hardware is trusted
usual assumption taken in system security field
like MMU's, disk encryption, ...

operating system (OS)
shares hardware between applications (CPU, memory, peripherals)
allows central mediation, is flexible & scalable
but full of bugs (30mio LoC, estimate of 15 bugs / 1000 LoC)

trusted computing base (TCB)
application itself (100k LoC)
OS (30mio LoC) & hypervisor (500k LoC)
BIOS & Intel Management Engine (unknown LoC)

starting applications example
user requests .exe to run
OS loads .exe & checks integrity
OS maps .exe to memory & sets up its own page tables
OS sets up IOMMU to protect against DMA access
OS starts execution

hardware assistance
CPU has priviledge rings, MMU
chipset provides DMA remapping tables
TPM & OS-enforced HDD access

physical attacks
hard to defend for OS
like remove hard drive, USB dongle boot
BIOS protection broken (reset using jumpers / removing battery)

paging-based security
supervisor bit (determines ring-0 access) isolates OS from applications
RW bits determines read/write of pages
execution bit (determines if executable) prevents run-time code injection
implemented by MMU, IOMMU

6.3 partial/full disk encryption

attacker cannot recover data / simply boot from USB
but can wipe disk, find out key via other ways

trivial disk encryption
user provides key to decrypt disk encryption key
disk encryption key placed in memory & used to decrypt disk
but could brute-force password

TPM supported disk encryption
use user-supplied key to unlock encryption key from TPM
as TPM can enforce trial wait-times
but requires hardware support & migrating data is hard

implementations
like bitlocker (windows), filevault (macosx), dm-crypt (linux)
different config (pw only, parts of encryption key in the cloud, ...)
disk only stores encrypted data
decryption / encryption using transparent layer

security guarantees
encryption key must be kept in memory
hence assumption that attacker can not read secret from memory

DRAM cold boot attack
can insta-freeze RAM (-50 °spray)
then plug it in another machine & read out contents
after 5s all OK, after 60s still large parts visible
the colder, the slower data decays

prevent cold boot attack
erase keys from memory (but sudden power loss problematic, bad UX)
prevent external booting (but can still transfer components)
physically protect against the cold / enclosure opening (but expensive)
avoid placing the key in memory (but requires architectural changes)

6.4 launch time integrity

use chain of trust
each before measures (checks integrity) of next application
BIOS → boot-loader → OS → application
as BIOS/boot-loader do not have driver to HDD need TPM

secure boot
OS only boots if chain of trust valid (OS signed by authority)
can load intermediate bootloader for other OSes
supported by BIOS and UEFI (replacement of BIOS)
use TPM to measure secure boot & attestate to it later

6.5 smartphone storage protection

users can enable storage encryption
encrypt with user-provided PIN; ARM TrustZone verifies PIN
trottling by CPU to prevent brute force
but failure counter stored outside chip in non-volatile memory (VRAM)
can mount replay attack possible using NAND mirroring

device specific encryption keys
PIN derives encryption key from PIN and processor key
storage can only be decrypted on same device as processor

NAND mirroring
reply (old) failer count message to prevent increasing wait times
backup NAND, try out PINs, restore backup on NAND, repeat
like possible with iPhone PIN protection

7 trusted execution environments (TEE)

put some trust on platform (chips, hardware)
like CPUs, use GPUs, use CPU + selected peripherals, ...
but without relying on too many peripherals
as alternative to OS-based security (replicates OS functionality)
for example useful for client wanting to attestate server process

7.1 target properties

isolation
isolate memory/storage (applications's data has confidentiality, integrity)
but OS should still manage memory, scheduling, peripherals

attestation
additional to isolation
remote party needs to know with whom it communicates

sealed storage
after local root of trust is setup successfully
enable local execution & fetching of secret data

7.2 implementation design decisions

isolation with virtual/physical memory
virtual memory is flexible, full usage of memory
but more attack surface (incl. side channels)
physical memory is simple, clean separation
but not flexible & some memory is wasted

resource management by OS
use OS to keep managing (virtual) memory, scheduling, peripherals
but have to protect applications memory (confidentiality, integrity)

minimal TCB
seperate security/privacy-sensitive parts of application
run only sensitive in enclave

rollback attacks
have to prevent simply resetting memory & restarting enclave
for example using monotonic counter

7.3 secure system

want to remotely archive secure code execution
while part of the system is untrusted
evaluation metric is size of TCB (trusted computing base)
hence what we need to trust for mechanism to work

security properties
ensure no screen / keyboard / webcam / micro recording
ensure code integrity, correct code run, memory protected

hardware-based attacks
add malicious chip (like enabling non-protected CPU mode)
use x-ray to check all gates in chip are expected
already occurred in practice (The Big Hack by china)

7.4 secure system approaches

read only memory (ROM)
keep entire program code in ROM
simple, adversary cannot add software
but cannot update (no bugfixes / new features)
but TCB is entire system (no isolation)
but control-flow based attacks possible (like ROP)

secure boot
load only code with valid signature
ensures only approved software is loaded
but TCB is entire OS
but undefined what exactly is executing

virtual-machines (VM)
virtual machine manager (VMM) launches VM for each application
can proof VMM to be correct & isolation enforced (as small)
but VMM has less features as real OS
but interaction between applications difficult (like clipboard)

signed code
only execute signed code
but new vulnerabilities might be discovered
need version hash-chain (to prevent running unpatched versions)
but signing key could be compromised (need certificate revocation)
need correct time (to prevent instantiation of old signature)

7.5 secure systems with attestation

enable verifier V to verify what is executing on untrusted device
V compares measurements with database of expected software
need some initial trusted system communicating with verifier
useful for OS, applications, firmware, ...

observations
need compatibility with buggy, insecure legacy software
hence try to archive security only for secure subset

general approach
establish isolated execution environment (partition from untrusted)
externally validate correctness (using some internal root of trust)
start autonomous operation through the validated environment

adversary model
controls network, compromises OS/applications
some minimal physical attacks (reboot, malicious USB-devices)
assume local hardware to be trusted (no state-level attacker)
assume no strong physical attacks (no firmware/hardware changes)

local attestation
cannot verify local running program (as verification can be faked too)
instead could place private key in secured area
app only gets useful result if private key used to decrypt

external attestation
need local root of trust to perform measurements
want smallest possible trusted computing base (TCB)

7.6 AMD SEV

uses AMD security processor core (exists in addition to normal cores)
protect VMs from untrusted hypervisor & other VMs
requires no changes in guest software

secure encrypted virtualization (SEV)
transparently encrypts VM memory content
with keys unique per VM; never visible to software/other hardware

properties
encryption-based isolation of virtual machines (confidentiality)
hypervisor continues to manage page mappings (hence no integrity)

secure nested paging extension (SEV-SNP)
reverse map table (RMP) performs checks on memory access
provides integrity protections against data corruptions, aliasing, replay

encrypted state extension (SEV-ES)
VM registers are encrypted & integrity protected (upon context switch)
helps to prevent exfiltration, control flow, rollback attacks

7.7 Risk-V KeyStone

Risk-V is Open Source architecture; KeyStone TEE extension
separates different priviledged modes

separates trusted & untrusted execution environments
trust assumptions include security monitor

modes (high to low)
M-mode (silicon root of trust, keystone security monitor)
S-mode (untrusted OS, trusted enclave runtime)
U-mode (untrusted OS applications, trusted enclave applications)

architecture
enclaves manage its own memory with PMP
enclaves can use formally verified enclave runtime

silicon root of trust
fundamental trust assumption
bootloader, keys, crypto engine, randomness, tamper-proof storage
measures/signs security monitor

physical memory protection (PMP)
configurable in M-mode
defines r/w/x for each mode / physical pages
only permission bits enabled for currently running OS / enclave

security monitor (SM)
like a hypervisor (manages memory, starts OS)
manages enclaves (OS helps, but loses access after start) & PMP entries
does remote attestation (measurements) of enclaves
upon enter/exit enclave, inverts PMP permission bits (except shared)

usage
user over untrusted network asks for application to be run
security monitor measures & signs enclave application

limited functionality
security monitor is small (only 10k LoC)
hence only limited functionality (context switch, create/destroy enclave)
OS does heavy lifting (like scheduling)

7.8 ARM TrustZone

widely deployed ARM hardware-assisted security
few reported attacks & vulnerabilities (in 2 decades!)
processor runs either in normal / secure world
secure applications must trust each other
trust assumptions include ARM

architecture
enclaves run on secure OS
no isolation between enclaves

Normal vs Secure World
normal world with (rich) OS/applications
secure world for protected/secure OS/applications
user/privilege mode of execution exists in both worlds

memory protection
trust zone aware address space controller (TZASC)
adds bit if memory belongs to secure/normal world
trust zone aware memory adapter (TZMA)
cache-level extension to remember world bit
separated MMU translation tables
read/write permissions by each world

device access
trust zone aware protection controller (TZPC)
which controls access, handler priority, ...
priorization prevents DoS by normal world

Cortex-A (smartphones)
many optional features; manufacturers choose what to implement
memory partitioned into normal / secure world w/ overlaps
NS bit determines in which world CPU is currently active in
enter secure word with interrupts
or calling secure monitor (using priviledged CPU instruction)

Cortex-M (microcontrollers)
design for low power / real-time (faster context switching required)
memory partitioned in secure / normal world
state of CPU depends on location of instruction pointer
enter secure world using the secure gateway instruction (SG)
SG jumps to non-secure callable (NSC), which jumps into secure world
avoids context-switch performance penalty with monitor mode (Cortex-A)
leave secure world with interrupts or BLXNS, BXNS instructions

7.9 ARM TrustZone-based security services

provides a trusted execution environment (TEE)

design choices
provide single TEE service with all functionality
has small TCB, but not flexible
provide TEE kernel supporting trusted apps
used in practice, but bigger TCB

kernel architecture
untrusted application passes pointer with input
untrusted OS calls trusted OS' secure monitor
trusted OS (TEE kernel) invokes trusted application
trusted application passes output to untrusted

kernel requirements
abstraction layer for convenient application development
services like secure storage for keys, crypto primitives
device access to fingerprint, SIM card, ...
message passing & sanitization
isolation between "untrusted" trusted apps
or allow only few "trusted" trusted apps from reputable sources

service examples
foundation (attestation & secure boot)
system services (disk encryption, key store)
third-party (OTP generator, payment, DRM, SIM lock, ...)

implementation
need careful software engineering & best practices
need hardened message passing interfaces
to avoid trivial side channels / API abuse
manufacturers usually only allow own applications

7.10 ARM TrustZone-based mechanisms

using the TEE, can implement various mechnisms

mechanisms
secure boot for brand / OS protection
full disk encryption to protect user data
hardware-backed key store to improve third-party security
device identification for regulatory compliance
smartphone as security token for second factor authentication

secure boot
for brand protection / safety settings (like radio) / app containerization
boot in secure world (integrity protected; like from ROM)
verifies boot loader (for example using ROM public key)
then boots normal world (for which boot loader verifies integrity)

full disk encryption
protect user data from theft
derive disk key from secure world key & password
rate-limit password guessing attempts
but replay attacks if non-volatile memory might be off-chip

hardware-backed KeyStore
provide API for apps to store secrets (key never leaves hardware)
but needs access control (user confirmation, rate limiting, expiry date)
else arbitrary content can be signed

attestation
verify code run in trusted mode ("secure provisioning")
need secured device-specific key & vendor certificate
verifier sends nonce to TEE kernel
trusted app supplies public key pk to TEE kernel
TEE kernel signs nonce, pk & infos about kernel & trusted app
verifier checks OK, then uses pk to encrypt input data for trusted app

trusted UI
include UI / touchscreen drivers
user directly communicates with secure world
but user might not be able to detect imposter UI

Sanctuary TEE architecture
sanctuary runs in secure world, starts enclaves in normal world
enforces isolation between enclaves & other applications
access enforced using coreID & current world of core
sanctuary provides remote attestation / sealing
but malicious cores can DoS / spoof coreID

7.11 ARM TrustZone security analysis

TEE kernels are non-trivial (vulnerabilities have been found)
off-chip memory allowed (to be able to increase TEE size)
only two security domains (vs SGX with many)

memory configurations
only on-chip (good physical protection, but limited TEE size)

with off-chip memory (TEE size increased, but physical attacks easier)

message passing
untrusted app provides pointer to input of trusted app
trusted app works on shared memory to improve performance
but secure world access to all memory can be dangerous

scripting for safer trusted apps
small interpreter for trusted apps written as scripts
very small footprint, allows third-party app development

7.12 ARM TrustZone attacks

boomerang
untrusted app passes malicious pointer as input
trusted app has no context about memory (might decrypt kernel data)

side-channels
needs secret-dependant branching
then abuse for example cache access patterns
but adversary interrupts must be precise & frequent (& then still noisy)

fault-injection
introduce fault which skips security critical check
like voltage manipulation to create hardware faults in secure world
but needs precise manipulation, suitable target code

7.13 Intel software guard eXtensions (SGX)

execute sensitive code in enclaves on hardware protected memory
user-space can request to call enclave
only encrypted / integrity protected traffic leaves processor

trust assumptions
trusted are intel, CPU, quoting enclave, intel SGX trusted libraries
untrusted is BIOS, firmware, OS, other software / hardware

properties
execution confidentiality (memory used by enclave encrypted by CPU)
code integrity (application binary signed)

remote attestation
assure enclave runs correct code on genuine platform & is untampered
V sends nonce
enclave generates report (= hash (code, data, stack, heap))
quoting enclave verifies report & signs with platform key
V verifies report & signature

attestation variants
needs communication with intel server to verify keys
enhanced privacy ID (EPID) which uses unattributed key
pseudoanonymous which generates new id, follow ups use same id
data center attestation primitive (DCAP) with 3rd party server
DCAP with self-rolled PKI

SGX implementation
runs on single code, microcode enforces isolation
DRAM memory is encrypted

virtual memory
want enclaves to use virtual memory for efficiency/flexibility
but prevent OS from accessing enclave memory / changing its mapping
hence duplicate all virtual memory in CPU

processor reserved memory (PRM)
memory protected from non-enclave memory accesses
enclave page cache (EPC) for pages that store enclave code & data
enclave page cache metadata (EPCM) stores physical <-> virtual relation

SGX create enclave
OS prepares memory region & loads code
hardware validates enclave using app certificate (=hash (app, client pk))
hardware generates enclave key K to store data in memory
enclave is started

conclusion
MMU-based isolation flexible, reuses existing mechanisms
but size of enclaves limited
but have to reimplement applications (with side-channels in mind)
several side- / covert channel attacks
limited size of enclaves

attacks
rollback (use old valid answer of SGX if no nonce used)
vulerabilities in SGX APIs
memory/cache based side-channels like page faults
speculative execution in concurrency (foreshadow)

7.14 trusted platform module (TPM)

widely deployed (100's mio devices, $1 cost)
relatively established (older than SGX)
passive device with specific operations (no general processor)
specification by non-profit trusted computing group (TCG)

applications
secure boot (system only starts if chain of trust valid)
authenticated boot (system records chain of trust, but starts always)
crypto co-processor (key generation, encryption)
secure storage (non-migratable keys)
trusted log (hash-chain based log)
attestation (proving system state with trusted log)
but poor performance

core goals
platform identity (prevent sibyl attacks)
remote attestation (incl. BIOS, OS, applications)
sealed storage (for secrets; only unseals if in expected state)
secure counter (only increments, prevent rollback)
secure random number generation
store storage root key to for disk/memory encryption

architecture
attached to the LPC bus (like CPU fan); slow but cheap
TPM certificates, non-volatile storage with private keys
platform configuration registers (PCR)
monotonic counters, secure random generator, key generation
hash generator, crypto RSA (encryption, signatures)

integrity measurement chain
measure OS/applications/config by hashing memory
core root of trust for measurement (CRTM) is BIOS
hashes chained togethers; head stored in TPM, rest in OS
verifier validates that head contains expected value
as TPM non-resetable, attacker needs to find collision / reboot

hash security requirements
attacker must not be able to find x such that
H(invalid || x) = H(valid) = expected hash value
hence weak collision resistance required

limitations
not tamper proof (too low-cost)
can read out private key with electron microscope (around $100k)
not within CPU (flash needs different manufacturing, liability concerns)
LPC bus make eves-dropping / inject attacks low-cost

possible attacks
compromise TPM directly (like malicious admin of cloud hosting)
compromise BIOS faking measurements
compromise vulnerable devices (like tpm.fail)

7.15 TPM attested/measured boot (TCG 1.1-style attestation)

also called static root of trust for measurement (SRTM)
hash all software & config which is executed
then remote verifier can validate measurements

SRTM security properties
measure entire system starting with immutable block
⇒ entire system part of TCB

process
trusted firmware hashes BIOS & loads it
BIOS hashes boot loader & loads it
then OS, then application, ...
PCR collects all hashes, signs & sends to the verifier
verifier compares with whitelist

integrity measurement architecture (IMA) by IBM
whole system is measured "on-demand"
measurement by already measured content of next executable
BIOS stores in PCR01-07
then bootloader stores in PCR04-05, at the end in PCR08
then OS stores in PCR10

linux modifications
measure linker, executable, shared libraries, kernel modules
measurement cache to measure unmodified files only once
find measurements in /proc/tcg/measurements
find PCR values in /proc/tpm/pcrs; PCR10 with result of measurements

use-cases
verify all devices in corporations run same software
verify user / cloud server runs expected software

anlysis
much better than antivirus
but still weak properties (no guarantee of running system, large TCB)
as measurement only at load-time, reset attacks & ROP still possible
due to many components (100s) large TCB & huge whitelist

7.16 TPM dynamic root of trust / late launch (TCG 1.2-secure operation)

also called dynamic root of trust for measurement (DRTM)
create isolated / trustworthy execution environment (IEE / TEE)
within untrusted OS, only small TCB needed
using CPU instruction, possible even after insecure boot

DRTM security properties
create execution environment
perform remote verification / attestation
establish secure channel
verify input I on code S produced output O within IEE

create IEE
SKINIT (AMD) and SENTER (Intel) to start IEE
soft-resets CPU, resets dynamic PCRs
setup DMA protection for code execution memory
secure loader block (SLB) hash stored in TPM
then execution SLB is started
atomic instruction needs around 1s to complete

simulate SKINIT
not possible, as it sets SCR 17 - 20 to 0
initial value at boot was -1
hence can detect if SKINIT was not called

verify / attest IEE
verifier sends nonce N, which is relayed to SLB
SLB attestates (signs over N and identity of SLB)
response relayed back to verifier

secure channel
include public key in response of SLB
verifier can then encrypt under public key

verify computation
pass nonce N and input I to SLB
SLB signs N, I, output O

security properties
similar to reboot; can late-launch secure kernel at run-time
small TCB (only hardware & application; no BIOS / bootloader / OS)
but memory unencrypted (local physical attacker unprotected)
but heavy-weight (1s system pause, no concurrency supported)
but no virtualization possible (else need to trust hypervisor)

7.17 DRTM flicker system

execute secure application then give control back to OS

properties
attests arguments, execution & protection of sensitive code
only 250 LoC (which is consequentially added to TCB)
invocation takes 1s, hence not very performant

process
app calls module in OS to start invocation
module stores OS state in RAM, then executes SKINIT
CPU resets dynamic PCRs, then starts flicker shim execution
shim starts executing secured app S
secured app delivers result to shim
shim restores OS, pass output of S and terminates

alternative TrustVisor
minimal hypervisor (only 7k LoC), 7% overhead
replaces expensive TPM functionality in software

7.18 TPM secure channel establishment

local party (LP, trusted) connects to remote host (RH, untrusted)
using a secret, authentic channel wants to execute with IEE

challenges
OS on RH is untrusted
IEE needs to be invoked several times (as OS does network)
OS stores state of IEE

assumptions
LH knows public key of TPM on RH
hardware is uncompromised (no bus snooping and similar)
strong random number available (to create secret key)
SKINIT extends PCR17 with H(IEE)

operations
extend(PCR, value) → PCR = H(PCR, value)
seal(PCRs, payload) → stores payload with PCRs & their values
unseal(PCRs) → returns payload sealed with same PCRs & value
quote() → signs PCR values & TPM arguments with TPM certificate

protocol I (RH generates RSA key)
LP <-> App (untrusted) <-> IEE
LP sends nonce to IEE
IEE creates pk/sk, seals under PCR17/18 (= hash of n, pk)
IEE extends PCR17/18 by stop
App quotes PCR17/18 and n
LP verifies PCR17/18, creates session key K encrypted under pk
IEE recovers secret key to get session key K

protocol 2 (LP generates RSA key)
LP <-> App (untrusted) <-> IEE
LP creates pk/sk and sends pk to IEE
IEE creates session key K, seals under PCR17/18 (= hash of pk)
IEE extends PCR18 by K, PCR17/18 by stop
App quotes PCR17/18
LP verifies PCR17/18, decrypts K using sk

protocol 3 (diffie hellman)
LP <-> App (untrusted) <-> IEE
LP creates g^r mod p = R
IEE creates K=H(R^s mod p), seals under PCR17/18 (= hash of R)
IEE extends PCR18 by g^s mod p, PCR17/18 by stop
App quotes PCR17/18
LP verifies PCR17/18, creates K with S

7.19 TPM cuckoo attack

valid attestation only guarantees some TPM was involved
but can MitM attacker-owned (and powned) machine

defense
have authentic key of TPM (but circular dependency)
display key in BIOS (but malware could fake screen)
seeing-is-believing (SiB) with barcode on physical device
have physical port (like USB) to retrieve TPM pk

7.20 TCG 1.2 for the cloud

as of 2020, not widely deployed
as lack of industry support, customer demand & weak properties

approach
inventory with public key for each device's TPM
customer is allowed to (randomly) select physical machines
surveillance for existing hardware so can not be tampered with

remaining challenges
enable migration, backup, recovery of secret keys
prevent single malicious administrators from stealing secrets
enable software updates without losing sealed keys
disable rollback to vulnerable versions / to reset state

7.21 comparison

trust assumption
on hardware manufacurers
with AMD/Risk-V/SGX unencrypted traffic never leaves CPU
vs TPM, which has to relay additionally on BIOS

isolation
by controlling memory access using dedicated hardware
vs TPM, which provides only secured execution on chip
vs TrustZone which allows off-chip memory

attestation
using hardware certificates (more or less hard to extract)
SGX needs intel to verify its certificates or different setup

runtime
KeyStone provides run-time
TrustZone provides secure OS
SGX provides secure enclaves
TPM only provides selected functionality

7.22 unique technology features

KeyStone

enforcement at the core
illegal access never reaches the bus
access policies needed at each core, synchronization required

TrustZone/Sanctuary
enforcement by memory controller
malicious cores might flood bus with illegal access / spoof master's ID

SGX
memory encryption (main contribution; no more eavesdropping)
execution can be unprivileged, multi-threaded, interrupted, combined with untrusted code
memory-intensive application faster due to cheap context switches
but side-channels (non-volatile memory access patterns, concurrency)
but less available, manual implementation of sealed storage/counters/attestation
slow counters (vs TPM)

TPM
more widely available on more systems
sealed storage, roll-back protected counters
simple attestation without on-line third party
but LPC-bus tampering (no encryption makes MitM practical)

8 software-only root of trust

attestation without pre-shared key / public key
useful on legacy devices / to support HW attestation
like securely reading out TPM public key

8.1 potential approaches

past human examples
abacus (verifiable how marbles move)
balance (verifiable how balance changes)
punchcard mainframe (sounds relate to executed code)

ROM only execution
only allow software stored in ROM to execute
but need to exchange ROM to update software
but still malware with gadget abuse possible

ROM bootloader
loads software to be executed, reboots each time
but impractical
but reboot can not be remotely verified

hash engine
computes hash of software & sends to verifier
but hash system could be faked

secured multi-party computation
whole computation encrypted (including input/output)
needs no trust into specific machine
but very expensive (1mio slower in general, 1k when optimized)

8.2 setting

untrusted device D, trusted verifier V
D executes verification function VF on memory
V decides upon output if memory indeed as expected

strawman verification function
V sends checksum request
D responds with hash value of memory
but attacker can precompute / replay correct hash value

strawman verification function with nonce
V sends checksum request with nonce
D responds with hash over nonce + memory region
but attacker can compute over expected memory content

reflection
fill memory with random content (use PRG with seed = nonce)
clear system state / disable interrupts
return hash over entire memory & system state
V checks duration, hash & system state
good idea because hard to predict/monitor processor behaviour
but attacker might still be able to simulate system state

genuinity
approach to implement idea of reflection
verification function does randomized memory access
V receives measured cash misses / hits & other parameters
difficult to simulate for adversary due to complex architecture
but verifier still needs to know / simulate microarchitecture

alien vs quine
externally measure time for each executed instruction
proposed to put secure loader with read/write commands
give command & measure duration until execution
keep rebooting for new commands
but only for slow systems (due to measurement precision)

8.3 swatt

verifier function does pseudorandom memory traversal
if attacker has to spoof memory, results in detectable overhead
but code must be optimal (which might be provable)
but no algebraic shortcuts (no caching / precomputation)
but verifier needs to know hardware (particularly clock speed)
but response could originate from different device (MitM, proxy)
but attacker could change hardware (to be faster)
hence attacker model supports software only

process
verifier sends nonce
verifier function does pseudorandom memory traversal
verifier checks checksum & computation time

swat implementation
generate RC4 character to get target memory address
load byte from memory & apply transformation
incorporate output into checksum
relatively simple (16 instruction / 23 cycles in inner loop)

swatt advantage
very few statements; additional attack statements are expensive
+13% (3 cycles) overhead for single if statement
as inner loop run many times, overhead likely measurable

checksum only over partial memory
trivial implementation computes checksum over whole device memory
but dynamic data (stack), secrets, protected regions
hence design verification code only checking parts
introduces new attacks ("memory copy attack")
hence need to include program counter & data pointer into checksum
but reading program counter slow (increases attacker advantage)

ICE assembly code
even simpler alternative for swatt; includes program counter
generate pseudo-random number with T function
(two mov, bis, add; very fast but secure)
load byte from generated memory location
incorporate this + program counter into checksum
but high overhead

8.4 pioneer

on complex architectures, time measurements challenging
use and fill bottleneck so no unexpected perf. improvements possible
choose trace cache (limits to 3 micro-ops per cycle)

time measurements challenges
exec time non-determinism (OoO execution, caches, parallelism)
optimal code proof difficult due to complexity
DMA, interrupts, exceptions, virtualization attacks

approach
verify code integrity through SW-only attestation
setup untampered execution environment
execute code

parts
checksum code (for checksum over itself + hash function)
hash function (to measure integrity of execution environment)
execution environment (invoked with extenal input)

verification function
compute checksum over itself & hash function
set up untampered execution environment
hash function measures integrity of execution environment
target code is invoked with input

protocol
V sends nonce, input at t1
D executes checksum code with nonce and responds quickly at t2
V ensures t2 - t1 is acceptable (else attacker could have forged)
D executes hash function with nonce over target code
V checks if hash valid (else execution environment tampered)
D executes target code with input

checksum forgery attacks

memory-copy / data substitution
code optimization & parallelism / superscalar architecture exploitation
pre-computation / replay attacks

execution tampering attacks
run malicious OS/VMM at higher privilege level
get control through interrupts & exceptions
but results in slowdown of execution

replace interrupt/exception handlers
to circumvent attacker-owned handlers
requires replication of drivers (as OS handlers replaced)
to replace, must check handler base address (which is slow)
in init code too early (attacker can jump over code)
in checksum code too slow (increases attacker advantage)
after checksum code too late (attacker can prevent real execution)

checksum over stack trick
place intermediate checksum results on stack
after checksum precomputation, handlers are replaced
then checksum finalized
if exception raised before replacing handlers
then temporary results on stack will be destroyed

results
archives code & launch point integrity (not control-flow integrity)
feasible even over network (experiment attacker disatvantage 0.3ms)
adresses many issues with moden CPUs / portability / ...
but needs protection against proxy / overclocking

8.5 shared secret in sensor network

many small devices form network
attacker controls memory / code (including keys) of compromised devices
in general impossible due to MitM

assumptions
attacker cannot compute faster
node has unique, public, unchangeable identity
local secure random number source

ICE key
use ICE to compute checksum
use as short-lived shared secret to authenticate diffie hellman
any device can compute this, but real device around 20% advantage

challenges
cannot do simple DH due to MitM
have to minimize computations to avoid attacker advantage

guy fawkes protocol
A/B pick random v_2 / w_2
A/B create hash chains $v_1 = \mathrm{H}(v_2)$, $v_0 = \mathrm{H}(v_1)$
assume A/B know v0 / w0 of each other
A sends v1, M_a, MAC(v2, M_a)
B checks if H(v1) = v0
B responds with w1, M_b, MAC(w1, M_b)
A checks if H(w1) = w0
A reveals v2, B checks consistency
B reveals w2, A checks consistency
hence v0, w0 act like public key for authentication of M_a / M_b
around 1% advantage to others

ICE key establishment (apply guy fawkes)
A picks random a
A computes g^a, computes hash chain $g^{a'}$, $g^{a''}$
A sends $g^{a''}$
B uses $g^{a''}$ as challenge for checksum c
B selects random value w2, computes hash chain w1, w0
B sends w0, MAC(c, w0) (response of B very fast!)
A sends $g^{a'}$
B selects random b, computes g^b
B sends w1, g^b, MAC(w2, g^b)
A sends g^a
B reveals w2
hence prevents MitM if attacker does not control network

8.6 recent research

optimality of hash function
use horer's rule for time-optimal polynomial evaluation
evaluation can be used as a hash function over memory
with provable optimality secure initial state can be established

open challenges
architecture-independent verification
high time difference between attack/legitimate function
extend trusted execution environment to graphic card

9 IoT (internet of things) security

many complex use cases, diverse environments
some applications have non-expert users
lack of incentive to secure system (secrecy, privacy)
cheap devices with low-cost hardware / outputs (like no displays)

9.1 security

security through proximity
assume signal is honest if strong enough (=distance low)
but with appropriate antenna & amplifier can increase signal strength
for example 1km for bluetooth with right antenna
or send a drohne / RC car for drive-by

security challenges
constrained capabilities and resources
(limited power, memory, CPU, bandwidth, range, UI, code size)
diverse communication technologies
(requiring translation gateways, proprietary protocols)

9.2 stack

popular technologies
differences in (network, power, data rate and range)
WiFi (LAN, high, 1.3 gbps, 100m)
bluetooth (PAN, low, 2.1 mbps, 100m)
zigbee (pan, very low, 250kbps, 20m)
6LowPAN (PAN, very low, 200 kbps, 20m)
LoRa (LAN, very low, 0.3-100kbps, 3-5km in urban area)

IETF standardization target
interopability (converge towards unified IP-based stack)
lightweight (small memory / code footprint, stateless, pre-shared keys)
includes header compression (IP addresses only in first package of stream)

IETF proposed stack
HTTP → CoAP (constrained application protocol)
TLS → DTLS (datagram transport layer security)
TCP / UDP → UDP
IP → 6LoWPAN (low power wireless personal attack networks)

open IoT stack
constrained application protocol (CoAP) instead of HTTP
datagram TLS (DTLS); security like vanilla TLS
UDP to transmit data
IPv6 to specify source / target
RPL to route in low power / lossy networks (AES encryption/authentication)
6LoWPAN as adaptation layer for IPv6 / RPL
physical / mac IEEE 802.15.4 (AES encryption/authentication)

future targets
tiny devices will have to relay on reduced stack
but might gain enough computing power to run full IP stack

9.3 event-driven communication leaks

encryption not enough for security
as devices leak device identifiers / communication patterns

example ZigBee
ZigBee assigns shot address (clear text) during association
this address is sent (again in in cleartext) in further communications
WiFi lock / unlock visible in traffic pattern
same for locks, lightbulbs, ...

infer user activity from encrypted traffic
(0) sniff wireless communication
(1) identify manufacturer / device from MAC address
(2) detect device events & device state depending on signal patterns
like missing lightswitch activity hints nobody is home

existential leakage
when existence of single message implies real-world event
like parking spot sensor activity implies new car parked
need dummy traffic to hide exact timing of message
but might need too much energy

statistical leakage
when deviation from normal implies real-world event

like more activity in parking spot implies company event
need same transmission rate in each time interval
but might need too much energy, congest network

9.4 other leaks

even if no communication leaks & strong encryption
still able to extract information (sidechannels & other)

inference of speech in encrypted VoIP
variable-bit-rate compression compresses sounds with varying fidelity
length-preserving encryption saves storage space
but length of message now allows to infer size of plaintext
can reconstruct message into phones (vowels or consonants)
able to infer clear-text phonemes (words) with 45% accuracy
through segmentation & classification, language correction, word segmentation & classification

traffic analysis in web applications
typed character with shown suggestions leaks pressed character
size of gif image reconstructs investment allocation
bc GIF uses runlength encoding; same pixels are collapsed

VPN leaks
packages are not normalized (same size, normalized timing) or reordered
hence VPN does not prevent this kind of attacks

9.5 pairing

establish security association between two devices
without shared prior knowledge
association only known to these two devices (like shared key)

secure device pairing
need resilience against man-in-the-middle (MitM) attack
harder with wireless channels & resource constraints (UI, UX, hardware)
no perfect / functional solution yet

use out-of-band (OOB) channel
assumption no MitM attack possible on secondary channel
only used for authenticating small amount of data
like comparing digits by human (but not typing keys)
not meant to replace high-capacity in-band channel

OOB developments
1999 resurrecting duckling (physical touch)
2002 talking to strangers (infrared)
2005 seeing is believing (visual, using QR codes)
2006 loud-and-clear (audio)

human-perceivable OOB
using visual (Line-Of-Sight), audio, haptic, sensing
but user needs to care (be alert, pay attention, correctly identify)
like bluetooth low energy (BLE) providing two options
user types in code of other device (but PIN guessable)
user compares & verifies codes (but user might not really check)

physically constrained OOB
using signal range (RFC, ...) or medium (water, body)
but needs hardware to support required properties
like shaking two devices hold together / same background noise
but video might help to detect patterns

use only in-band (primary) channel
as requiring a secondary channel might be impractical
use physical properties / modulation techniques to prevent alteration
but inherently difficult to prevent MitM attack
like BLE methods

9.6 bluetooth low energy (BLE) authentication

authentication mechanism
just works (no authentication)
passkey (entering ping on another device)
numeric comparison (confirming same pin on different devices)
out-of-band (assumes shared key already secretly established)

just works problems
temporary key simply assumed 0, hence even passive attackers succeed
in newer protocol now DH used (hence need active attacker now)
but need to support old devices (dumb down attacks possible)

passkey entry problems
can be broken by passive attacker
establishes temporary key, but keyspace only 1mio
hence can bruteforce, only marginally harder than just worksliced

9.7 ZigBee Light Link (ZLL) Touchlink commission

to communicate between user (U) and lightbulb (L)

protocol
U sends scan request with nonce n
L sends scan response with n
U sends join request, network key nk encrypted with master key, n
L sends join response, status, n
U sends commands encrypted with nk

master key
secret shared by all certified ZLL devices
distributed to certified manufacturers
but can open memory hardware & extract key
9F 55 95 F1 02 57 C8 A4 69 CB F4 2B C9 3F EE 31

critical commands
using only master key, can send commands to lightbulb
will only trigger if signal strength about threshold
identify by blinking, define duration up to 18.2 hours
reset to factory settings
change network (channel) settings
set random network key

malicious firmware using update mechnism
standardized OTA update mechanism which verifies updates
should use async crypto, but philips uses symmetric key
can find out key using differential power analysis
make bulb join compatability network w/o proximity checks
can now update OTA firmware signed with extracted key

IoT worm
device is vulnerable using radio commands
but can itself sends radio commands
hence worm can propagate without third party
use percolation theory to estimate how many devices needed
assuming uniformity, then N = 1.128 ∗ A(=area) / (pi ∗ r =(range) ˆ 2)
attacks include bricking, hamming, in/exfiltration of data
classic defense techniques do not work (firewalls, airgapping, ...)

10 OT cyber protection

operational technology (OT) manages infrastructures
while hard to enter, damage causable significant
like BKW (produces & delivers energy)

10.1 cyber security basics

cyber security incident
loss/manipulation of view/monitor/control
leads to safety, productivity and quality issues
then reputation loss, financial loss, causalities

cyber activists
hacktivists (idealism)
vandals / hobby hackers (trophy-hunters)
nation states (political)
cyber criminals (economic motivation)
terrorists (political)
relevant in CH are nation states and cyber criminals

cyber security blueprint
goal is to gain visibility and protect from events
visibility (know what is in control network)
prevention (implement proactive controls)
continuous monitoring (ensure process functions as intended)

NIST cybersecurity framework
identify and design (risks, responsibilities, ..)
protect (safety, reliability, ...)
detect (anomalies)
respond (incident response)
recover (tested recovery)

10.2 IT-sec vs OT

IT-sec protects information
vs OT protects physical operations

corporate IT
availability and safety most important, then integrity and confidentiality
target is to keep it running reliably for long time spans
long lasting, "old" technologies
patches & equipment build for decades

OT
confidentiality most important, then integrity and availability
target is to improve productivity, keep CIA
chaining, evolving technologies
frequent patches & new devices

OT developments
randsomware attacks start becoming more frequent for OT
like 2020 honda devices taken down

10.3 OT security analysis

OT challenges
networks not completely air-gapped
updates expensive (system restart expensive, updates not supported)
favour on availability (vs security)

attack vectors
malware infecting human facing interfaces
malware circumventing built-in invalid input filtering
vulnerable industrial control exposed to internet
third party devices breaching network segregation
disgruntled employee breaches security
employee is victim of social engineering

challenges
many assets & configuration unclear
workflow automation missing

targets
create an inventory of devices & configuration
regularely patch systems
monitor network & devices for malicious activity

10.4 OT protections

purdue model
strict network segregation with firewalls in between
0 (process like motors, sensors)
1 (basic control like PLCs)
2 (area control like alarms, manufacturing area with single purpose)
firewalled to
3 (site control center like factory schedule, workstations)
separated with demilitarized zone (DMZ) from
4 (site business planning like emails, internet access)
5 (enterprise like connection to partners)
in DMZ, all data should end/originate (no pass-through)

sensor protection
add sensor control system to network
which makes it possible to locate attackers within network
central protection (single control system)
partial protection (control system per substructure)
full protection (each facility with own control system)

target sensor architecture
sensor added to switch
then OT monitoring platform produces logs
then collected and sent to SIEM (monitoring system)
then SIEM combines with other sources and sends to control
control reacts if anomalies detected

passive scanning
connected to SPAN port of OT switch
listens to network traffic
easy but not so accurate

active scanning
finds devices / open ports in network
derives agent type & collects PLC configuration
probe using native OT protocols, alter upon unexpected state
allows deep analysis, accurate asset inventory tracking
but expensive products as have to integrate OT protocols

hardware enforced protection
OT network n_{ot} has only unidirectional gateway into company network n_c
measurements within n_{ot} placed on replica database / devices in n_c
flip physical switch to allow requests back (like to update devices)
setup monitoring, support, cloud backups within insecure n_c
without opening security holes, as no requests into n_{ot} possible

smart meters
sensor results are aggregated and then sent to target
but attacker on path might change measurements or enter network

vulnerability monitoring
inventory of devices observed for vulnerabilities
critical software versions raise alarms

11 appendix

11.1 TPM secure channel details

protocol I (RH generates RSA key)
LP <-> App (untrusted) <-> IEE
LP generates nonce n and sends it
App relays n to IEE
IEE generates pk, sk
(need to generate pk, sk within IEE, else App learns it)
IEE extends (=adds) n and H(pk) to PCR18
IEE seals private key sk with (PCR17, PCR18)
so TPM can only reciver sk if same values in PCR17, PCR18
IEE extends stop value to PCR17, PCR18
so when returning to App, App cannot unseal
IEE returns pk to App
App performs quote (attestation) of n, PCR17, PCR18
App sends quote & pk to LP
LP verifies PCR17 contains H(0 || H(IEE) || stop)
LP verifies PCR18 contains H(H(H(0 || n) || H(pk)) || stop)
LP generates session key K
LP encrypts K with pk, encrypts command with K and sends to app
App passes both values and pk, n to IEE
IEE uses n and pk to unseal sk from TPM
IEE decrypts K using sk
IEE decrypts command using K
IEE may seal K (for next communications)

protocol 2 (LP generates RSA key)
LP generates pk, sk and sends the pk
App forwards pk
IEE generates session key K
IEE extends PCR-18 with H(pk) and K
(need to extend with K, else MitM by App possible)
IEE seals PCR17, PCR18 with K
IEE extends stop value to PCR17, PCR18
IEE encrypts K with pk and sends to App
App performs quote of PCR17, PCR18
App sends resulting signed hash and encrypted K to LP
LP decrypts session key K
LP verifies PCR17 contains H(0 || H(IEE) || stop)
LP verifies PCR18 contains H(H(H(0 || pk) || K) || stop)
LP encrypts command with K and sends to app
App passes this and pk to IEE
IEE uses pk to unseal K from TPM
IEE decrypts & processes command

protocol 3 (diffie hellman)
LP generates secret r, sends g^r = R to App
App forwards R
IEE generates secret s, computes g^s = S and K = H(R^s)
IEE extends H(R) into PCR18
IEE seals PCR17, PCR18 with K
IEE extends H(S) to PCR18
(need to extend to prevent MitM, after seal to avoid storing s)
IEE extends stop value to PCR17, PCR18
IEE sends S to App
App performs quote of PCR17, PCR18
App sends resulting signed hash and S to LP
LP verifies PCR17 contains H(0 || H(IEE) || stop)
LP verifies PCR18 contains H(H(H(0 || H(R)) || H(S)) || stop)
LP calculates K = H(S^r)
LP encrypts command with K and sends to app
App passes this and R to IEE
IEE uses R to unseal K from TPM
IEE decrypts & processes command

design considerations
generate all secrets within IEE (as all others untrusted)
include all secrets in PCR registers (else MitM possible)
use PCR-17 in seal op as this assured valid IEE code is run