Software Security in the Presence of Faults

Peter Gutmann

University of Auckland

Crypto Fault Attacks

If you get a fault during a crypto computation, an attacker may be able to recover your encryption key(s) from the faulty output

- First (publicly) acknowledged in the late 1990s
- Studied to death since then

Information Security and Cryptography
Marc Joye
Michael Tunstall Editors
Fault Analysis in Cryptography
Springer
Luca Breveglieri
Israel Koren
David Naccache
Jean-Pierre Seifert (Eds.)
LNCS 4236
Fault Diagnosis and Tolerance in Cryptography
Third International Workshop, FDTC 2006
Yokohama, Japan, October 2006
Proceedings
Springer
Stefan Mangard
Axel Y. Poschmann (Eds.)
LNCS 9064
Constructive Side-Channel Analysis and Secure Design
6th International Workshop, COSADE 2015
Berlin, Germany, April 13–14, 2015
Revised Selected Papers
Springer
Tim Güneysu
Helena Handschuh (Eds.)
LNCS 9293
Cryptographic Hardware and Embedded Systems – CHES 2015
17th International Workshop
Saint-Malo, France, September 13–16, 2015
Proceedings
Springer

Faults in Cryptosystems

ECC is particularly susceptible to faults

- Fault with the in-memory key: Leak the private key
- Fault with the ECC computation: Leak the private key
- Fault with the RNG: Leak the private key
- You get the picture

General idea is to move the computation from the secure curve to another, inevitably weaker, one or to produce a faulty point on the original curve

> Faults can be injected in a variety of ways and almost all parts of the system can be targeted, e.g. the base point, system parameters, intermediate results, dummy operations and validation tests
>
> — "Fault Attacks on Elliptic Curve Cryptosystems"

Faults in Cryptosystems (ctd)

RSA also has issues, but nowhere near as bad as ECC

RSA has a fault problem in the RSA-CRT computation if you sign the exact same message twice

- This essentially never occurs
- Not in IPsec, SSH, TLS, CMS, S/MIME, PGP, SCEP, TSP, OCSP, CMP, …
- Any protocol that allowed this would also trivially allow replay attacks

ECC in contrast has entire catalogues of fault problems

- These don't require duplicate signatures

Faults in Cryptosystems (ctd)

SRP, PSK, etc have no issues

- Authentication doesn’t require the use of signatures
  - Or certificates, or CAs, which is why there’s close to zero support for it in browsers
- Built around MACs/PRFs (hash-based)
- Little research published on the issue, but probably because there’s no obvious attack

Faults in Cryptosystems (ctd)

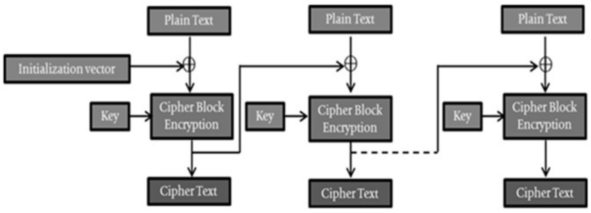
Symmetric crypto (e.g. AES) doesn’t have random fault issues

Attacks require injection of specific attacker-controlled faults, not random faults in random locations

- Example: Create 1-byte differentials in input to AES MixColumns
- Example: Create 255 different byte faults in the AES middle rounds
- Example: Create 1-bit fault in 128 bits of SubBytes input to AES last round

Faults in Cryptosystems (ctd)

Similar to RSA, attacks require encryption of the same data two or more times



Source: Microsoft

- Won’t happen for the common CBC or CTR/GCM modes

CTR/GCM mode, however, fails catastrophically on an IV fault

- Both confidentiality and integrity protection collapse

Faults in Cryptosystems (ctd)

The two trendiest encryption mechanisms, ECC and AES-GCM, are also the most brittle in the presence of faults

- Worst case, a fault in the RNG, and you lose everything in one go
- ECC private key
- AES-CTR confidentiality
- AES-GCM integrity-protection

The most robust mechanisms are probably RSA and AES-CBC + HMAC

- They’re not fashionable ☹

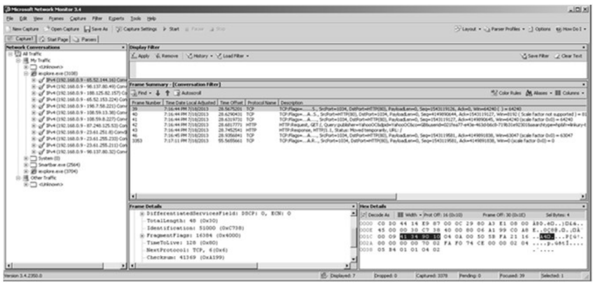
Faults in Cryptosystems (ctd)

How to carry out the attack

- Wait

Purely passive attack

- No need to do anything except watch network traffic and wait for it to happen



Frame Number	Time Date Local Adjusted	Time Offset	Protocol Name	Description
39	7:16:44 PM 7/18/2013	28.5675201	TCP	TCP:Flags=......S., SrcPort=1034, DstPort=HTTP(80), PayloadLen=0, Seq=1543119126, Ack=0, Win=64240 ( ) = 64240
40	7:16:44 PM 7/18/2013	28.6290431	TCP	TCP:Flags=...A..S., SrcPort=HTTP(80), DstPort=1034, PayloadLen=0, Seq=4149890644, Ack=1543119127, Win=8192 (Scale factor not supported) = 8192
41	7:16:44 PM 7/18/2013	28.6319731	TCP	TCP:Flags=...A...., SrcPort=1034, DstPort=HTTP(80), PayloadLen=0, Seq=1543119127, Ack=4149890645, Win=64240 (scale factor 0x0) = 64240
42	7:16:44 PM 7/18/2013	28.6817771	HTTP	HTTP:Request, GET /, Query:publisher=YahooOCBdpid=YahooOCBco=GB&userid=[illegible]
43	7:16:44 PM 7/18/2013	28.7452541	HTTP	HTTP:Response, HTTP/1.1, Status: Moved temporarily, URL: /
46	7:16:45 PM 7/18/2013	28.9356841	TCP	TCP:Flags=...A...., SrcPort=1034, DstPort=HTTP(80), PayloadLen=0, Seq=1543119581, Ack=4149891038, Win=63047 (scale factor 0x0) = 63047
3353	7:17:11 PM 7/18/2013	55.5655661	TCP	TCP:Flags=...A.R.., SrcPort=1034, DstPort=HTTP(80), PayloadLen=0, Seq=1543119581, Ack=4149891038, Win=0 (scale factor 0x0) = 0

```
DifferentiatedServicesField: DSCP: 0, ECN: 0
TotalLength: 48 (0x30)
Identification: 51000 (0xC738)
FragmentFlags: 16384 (0x4000)
TimeToLive: 128 (0x80)
NextProtocol: TCP, 6(0x6)
Checksum: 41369 (0xA199)
```

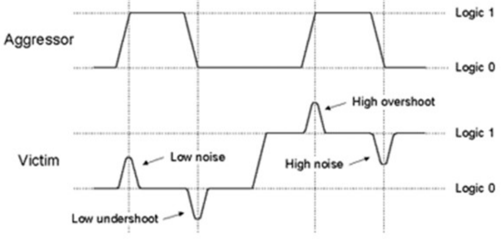
```
0000  C0 D0 44 14 E9 87 00 0C 29 80 A3 E1 08 00
000E  45 00 00 30 C7 38 40 00 80 06 A1 99 C0 A8
001C  00 09 41 34 90 10 04 0A 00 50 5B FA 21 16
002A  00 00 00 00 70 02 FA F0 74 CE 00 00 02 04
0038  05 B4 01 01 04 02
```

Source: Microsoft

Causes of Faults

Electrical glitches

- Overvoltage
- Undervoltage
- Spikes
- Clock glitches
- Noise



Source: EETimes

Thermal issues

Radiation

- Often induces electrical glitches
- Can also change circuit operation, temporarily or permanently

Characteristics of Randomly-appearing Faults

Possible: Random bit(s) 0 → 1

Possible: Random bit(s) 1 → 0

Unlikely: Random bit fault during computation

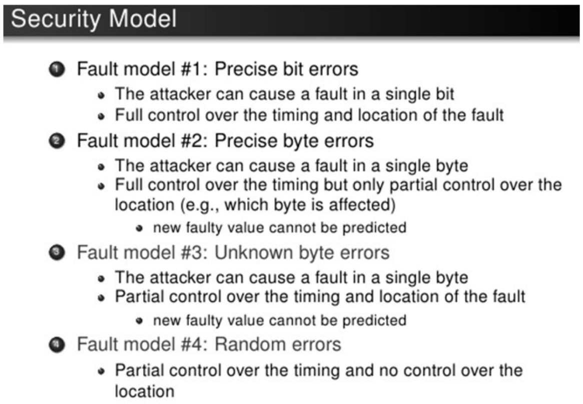
- Most CPUs have at least error detection on the CPU core
- Some have full ECC and more, e.g. Cortex A, Cortex R, IBM Power, Intel, MIPS, Sparc
- See later slides for extreme cases, e.g. Intel, IBM, Sparc

Not present: Non-random, attacker-controlled faults

- In any case if an attacker can disassemble your device and sit there injecting controlled hardware faults at will, it's probably game over anyway

Theory vs. Reality

Research results are often difficult to apply…



Security Model

1. Fault model #1: Precise bit errors
   - The attacker can cause a fault in a single bit
   - Full control over the timing and location of the fault
2. Fault model #2: Precise byte errors
   - The attacker can cause a fault in a single byte
   - Full control over the timing but only partial control over the location (e.g., which byte is affected)
     - new faulty value cannot be predicted
3. Fault model #3: Unknown byte errors
   - The attacker can cause a fault in a single byte
   - Partial control over the timing and location of the fault
     - new faulty value cannot be predicted
4. Fault model #4: Random errors
   - Partial control over the timing and no control over the location

Source: Gemplus

Fault model #5: No control over the timing or location, no duplicate data to act on

Theory vs. Reality (ctd)

Lack of understanding by cryptographers…

> release even _one_ any way faulty signature computed using RSA-CRT and your private key walks
> — CFRG list comment

- Garbled sound byte from a 20-year-old research paper

… or appreciation that TLS crypto exists outside the web…

> […] non-starter as web browsers […] fix the reasons why web browsers […] the web browser vendors […]
> — CFRG list comment, responding to a message that talked specifically about non-browser TLS use

Theory vs. Reality (ctd)

… or just plain denial

> I'm aware of invalid curve attacks, which can be completely mitigated by using a twist-secure curve and point compression
> — CFRG list comment

- "The mathematician looked at the fire extinguisher and the fire, said 'a solution exists', and went back to bed"

Cryptographers and SCADA/embedded implementers don't talk to each other

- Cryptographers: They're not using our fine theoretical design!
- Implementers: This stuff doesn't do what we need, we'll have to come up with our own way of doing it

Theory vs. Reality (ctd)

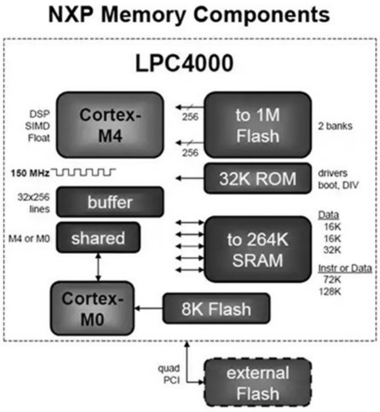
A few studies published, but all for code (not data) corruption

- 2% of firewall code-memory faults caused security problems
  — “Evaluating the Security Threat of Firewall Data Corruption Caused by Instruction Transient Errors”
- 1-2% of FTP and SSH code-memory faults caused security problems
  — “An Experimental Study of Security Vulnerabilities Caused by Errors”

Theory vs. Reality (ctd)

Code corruption isn’t normally an issue in fault-aware embedded systems

- Code executes directly out of nonvolatile memory
- If the code is in RAM, the RTOS monitor process scans the code segment and restarts the system on error



Source: NXP

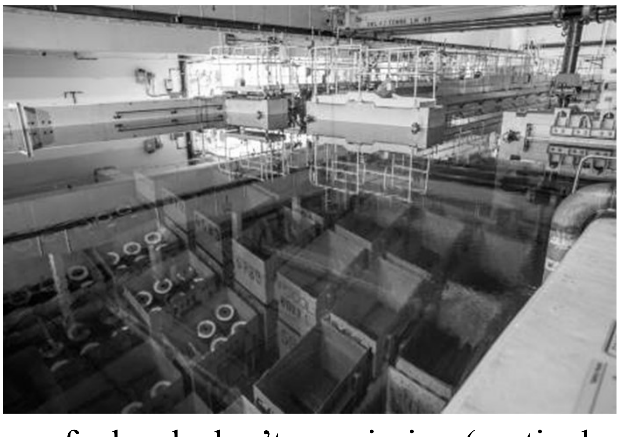
When are there Radiation-induced Faults?

When you’re using the crypto to monitor nuclear materials

Used to check compliance with nonproliferation treaties

Crypto in High-radiation Environments

Monitoring of fuel storage ponds



Source: World Nuclear News

Ensure fuel rods don’t go missing (particularly in breeder reactors)

Crypto in High-radiation Environments (ctd)

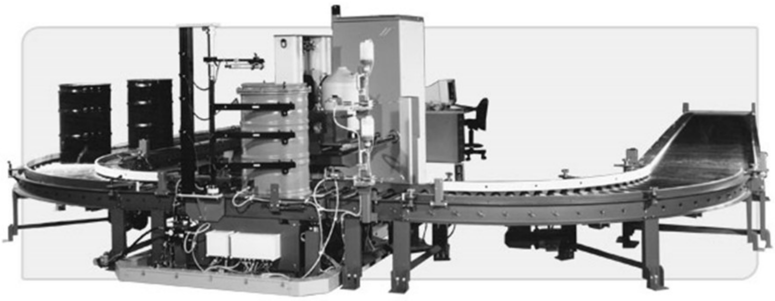
Monitoring of reactor refueling

Source: World Nuclear News

Check what goes in and out

Crypto in High-radiation Environments (ctd)

Monitoring of waste management



Source: Canberra

Check what's leaving the facility via nondestructive assay (NDA)

Crypto in High-radiation Environments (ctd)

Most of those aren’t truly high-radiation environments

- Humans have to work there

Higher-than-normal radiation, but not classed as high-radiation

- Other equipment is deployed to high-radiation areas

Leads to an interesting definition of tamper-*discouraging* crypto

> It would take you three days to put up the scaffolding and disassemble the monitoring gear. The radiation will kill you in one day

- Who needs “tamper-resistant” when you’ve got that…

Crypto in Harsh Environments

Not specific to reactors though…

Devices can experience faults in harsh environments in general

- Covered by numerous standards
- EN 50128 – Railway applications – Communication, signalling and processing systems
- EN 50129 – Railway applications – Safety related electronic systems for signalling
- EN 50402 – Requirements on the functional safety of fixed gas detection systems
- IEC 60601 – Medical electrical equipment safety

[Continues]

Crypto in Harsh Environments (ctd)

[Continued]

- IEC 60880 – Nuclear power plants – Instrumentation and control systems important to safety
- IEC 61508 – Functional Safety
- IEC 61511 – Safety instrumented systems for the process industry sector (also ANSI S84)
- IEC 61513 – Nuclear power plants – Instrumentation and control important to safety
- IEC 62061 – Functional safety of electrical, electronic and programmable electronic control systems (also ISO 13849)
- ISO 26262 – Road vehicles – Functional safety

Many, many more

Notable Failures due to Ionising Radiation

Advanced Simulation and Computing Program (ASC) Q Supercomputer at Los Alamos

Source: LANL

- Built with DEC Alpha 21264 CPUs

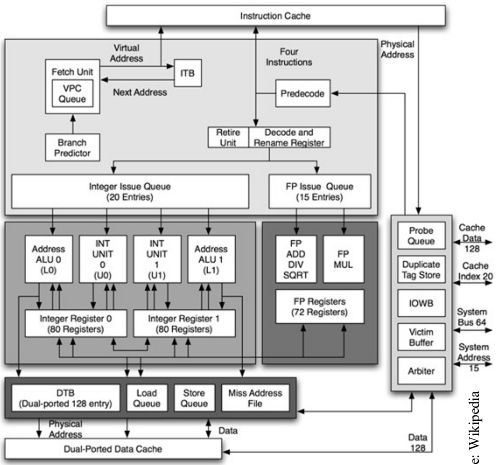
Notable Failures due to Ionising Radiation

Error detection but not correction on level 3 cache tag (BTAG) RAM

- Too much slowdown in this speed-critical case
- Standard data RAM does have ECC

Faults were detected via parity checks, but not corrected, node crashed

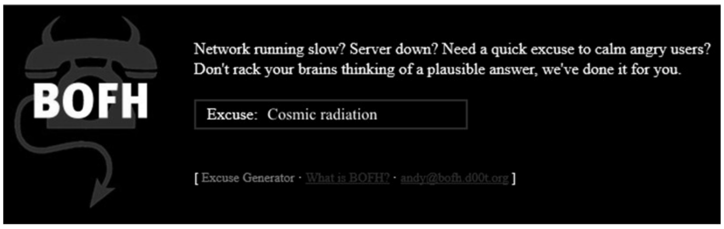
- c.f. IBM Power, which treats a cache error as a miss, not a fault



Source: Wikipedia

Notable Failures due to Ionising Radiation

Suspected cosmic rays



Network running slow? Server down? Need a quick excuse to calm angry users? Don't rack your brains thinking of a plausible answer, we've done it for you.

Excuse: Cosmic radiation

[Excuse Generator · What is BOFH? · andy@bofh.d00t.org]

Source: d00t.org

ASC Q is at LANL, elevation 7,500ft (2,300m)

- Cosmic radiation is 6x as intense as at sea level

For comparison, avionics computers are at 30,000ft

- Radiation is 150x as intense as at sea level

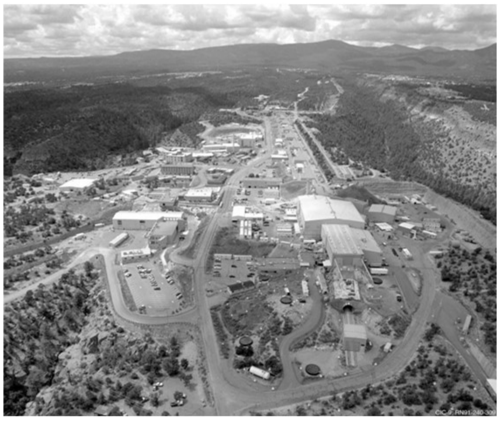
Notable Failures due to Ionising Radiation

Single node at sea level experiences fatal soft error once in 50 years

- 500-node cluster at elevation experiences one every 1½ hours

Los Alamos just happens to have the Los Alamos Neutron Science Centre (LANSCE)

- Confirmed that it's radiation-induced



Source: LANL

Notable Failures due to Ionising Radiation

Dealt with by

- Scrubbing cache RAM before program runs
  - Manual equivalent of automated ECC scrubbing
  - Rewrite ECC'd data with original correct data
- Checkpointing during runs to allow recovery
- Leaving spare nodes available to restart failed jobs on
- etc

(NB: Often-repeated 2016 IEEE Spectrum article mentions more examples, but these contain multiple factual errors and/or are unverifiable. Don't believe what Google will turn up).

Modern CPU Fault Resistance

Things can fail in unexpected ways

- Expose PIII and K7 to gamma source

Source: Intel

What failed wasn't the CPU but the CPU fan

- A PWM fan-control chip in the fan motor died long before the CPU did

Modern CPU Fault Resistance (ctd)

During the test, all components except the CPU were heavily shielded

- CPU was raised up above the shield by a riser board

Scattering caused faults in the shielded components

> Multiple motherboards, memory modules and video cards have lost functionality in the pursuit of the total dose limit of the DUT processors
>
> — "Total Ionizing Dose Testing of the Intel Pentium III and AMD K7 Microprocessor"

Modern CPU Fault Resistance (ctd)

PIII took 100x the dose of the K7

> Intel processors verge on radiation-hardened devices
> — "Terrestrial-Based Radiation Upsets: A Cautionary Tale"

- Possibly due to Intel's bad experience with alpha-particle induced faults in 16K DRAMs in the late 1970s

Use e.g. ECC and bit-interleaving (to fortify SECDED) in caches (c.f. 21264 BTAG issues)

Modern CPU Fault Resistance (ctd)

> It's clear that [Intel] are addressing an issue with cosmic rays, since they have become progressively more rad-hard over the years
> — "Terrestrial-Based Radiation Upsets: A Cautionary Tale"

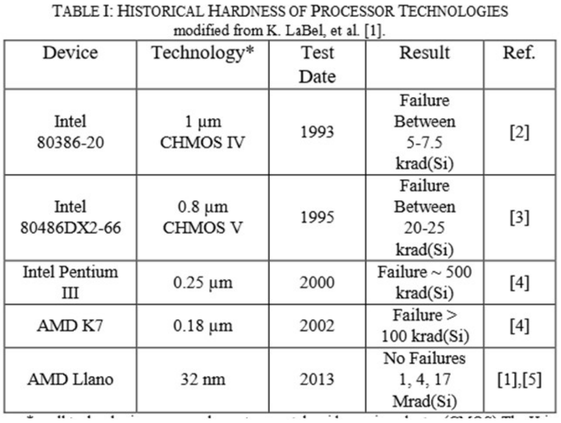


TABLE I: HISTORICAL HARDNESS OF PROCESSOR TECHNOLOGIES
modified from K. LaBel, et al. [1].

Device	Technology*	Test Date	Result	Ref.
Intel 80386-20	1 µm CHMOS IV	1993	Failure Between 5-7.5 krad(Si)	[2]
Intel 80486DX2-66	0.8 µm CHMOS V	1995	Failure Between 20-25 krad(Si)	[3]
Intel Pentium III	0.25 µm	2000	Failure ~ 500 krad(Si)	[4]
AMD K7	0.18 µm	2002	Failure > 100 krad(Si)	[4]
AMD Llano	32 nm	2013	No Failures 1, 4, 17 Mrad(Si)	[1],[5]

Source: Preliminary Radiation Testing of a State-of-the-Art […] Chip

Modern CPU Fault Resistance (ctd)

No apparent device degradation was apparent on any of the samples. Cumulative dose levels for exposures ranged from 1 to 17 Mrad(Si). For comparison, the ITAR level is 500 krad(Si).

As noted, total dose and DR [dose rate] device tolerances exceed the ITAR limits for this [AMD A4-3300, 2011 vintage budget desktop CPU] off-shore fabricated design. To the best of the authors' knowledge, AMD has not intentionally radiation hardened the device for these environments, but the technology itself supports these characteristics

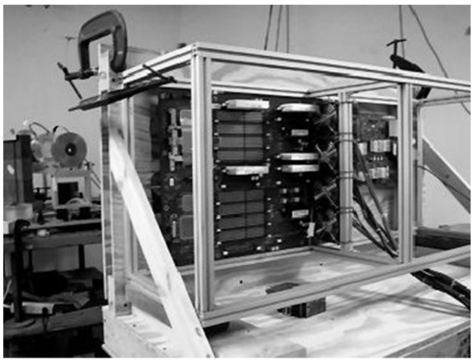
— "Hardness Assurance for Total Dose and Dose Rate Testing of a State-Of-The-Art Off-Shore 32 nm CMOS Processor"

- ITAR = International Traffic in Arms Regulations (now Wassenaar), who set the limits where something becomes an export-controlled rad-hard military device

Modern CPU Fault Resistance (ctd)

IBM Power is just as careful

- ECC, parity, residue checking, bit-interleaving ('chipkill'), automatic instruction retry, …
- Tested by proton-beam irradiation



Source: IBM

Modern CPU Fault Resistance (ctd)

Other server-grade CPUs like Sparc64 contain similar measures

- SECDED on level 2 caches
- Parity check on level 1 cache causes a reload from ECC-protected level 2
  - c.f. Alpha fail on parity error
- TLB also has parity check, error treated as a miss
- ALU has parity and mod-3 arithmetic checking of results
  - Failed instructions are restarted on error
- 10% of transistors are for error handling

Under intense neutron bombardment, 94% of errors vanished, 5% were recovered from, 2% resulted in an observable fault

Modern CPU Fault Resistance (ctd)

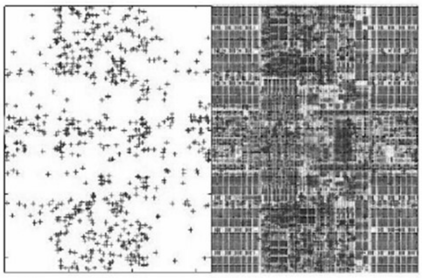
Rate of SEUs is measured in Mean Time To Upset, MTTU

Intel claim MTTU of 25 years for server-grade CPUs

- Presumably at sea level
- Possibly in a lead vault?

IBM claim "dramatic improvements in soft-error recovery"

- Compared to what?



As part of the process to verify the coverage model, the latch flip distribution (left) was overlaid on a POWER6™ die photo (right).

Source: IBM

Fujitsu (Sparc64) claim 10 FIT at sea level

- FIT = failure in time, per billion hours, MTTF of 10K years

Whole-System Fault Resistance

This is for $1,000 server CPUs

- And server-grade hardware in general, e.g. ECC RAM

Everything else isn't so seriously engineered

- Consumer-grade CPUs
- Embedded CPUs
- DRAM
- System buses
- I/O devices

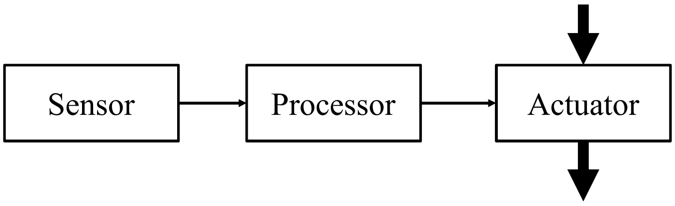
How do we build a reliable system from unreliable components?

Fault-tolerant Systems

Numerous architectures designed for functional safety

- General terminology is XooY
- X out of Y units must fail for a system failure

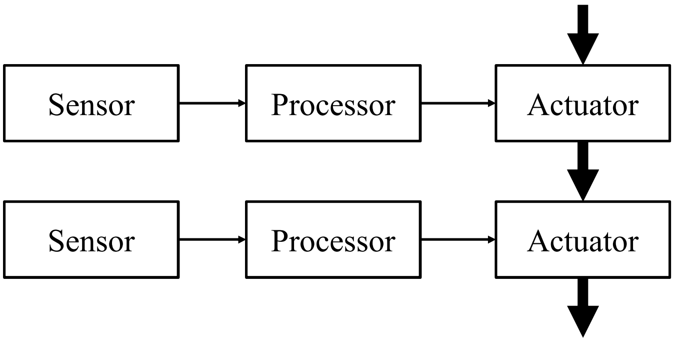
Standard systems are 1oo1



- Failure of any part causes system failure

Fault-tolerant Systems (ctd)

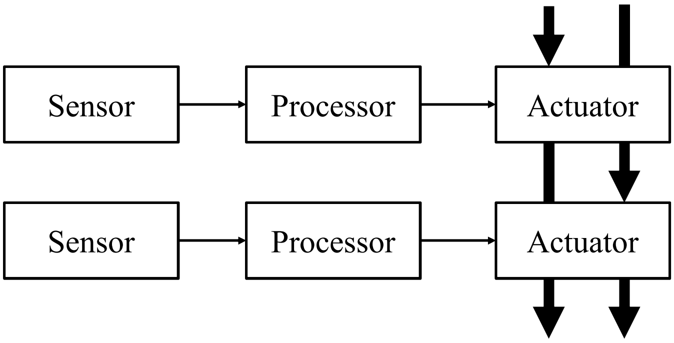
1oo2 for fail-safe systems



- Failure in both systems is required for an inadvertent activation to occur

Fault-tolerant Systems (ctd)

2oo2 for high-availability systems



- Both systems must fail for overall failure

Fault-tolerant Systems (ctd)

Even more complex systems are possible

- 2oo3 with voting circuits

All of these (except 1oo1) require custom hardware designs

- Not practical to require this
- Can't demand completely new hardware just to accommodate an obscure crypto issue, or even a less obscure security issue

None are really practical for general-purpose use

- May be feasible, but not really practical

Fault-Resistant Systems

There's a special variant that requires little to no custom work…

1oo1D

- Standard 1oo1 with diagnostic channel
- If a failure is detected by the monitoring system, halt or restart the main system

Fail-fast

- 1oo1D is pretty standard for radiation-tolerant systems
- Actually it's pretty standard for properly-designed (SCADA, not IoT) embedded in general

Goal: Make general-purpose software 1001D

Fault-Resistant Systems (ctd)

Swiss Cheese Model of Failure (Prevention)

- Developed by Prof.James Reason, “The Contribution of Latent Human Failures to the Breakdown of Complex Systems”

Widely used in

- Risk management
- Healthcare
- Engineering
- Aviation
- ...

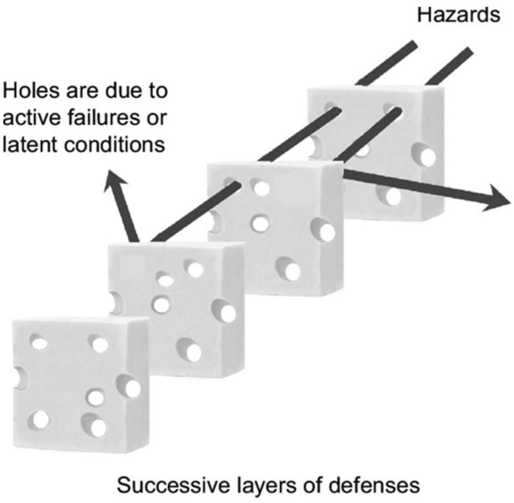
Fault-Resistant Systems (ctd)

Basic premise: Defences are imperfect

- Think Swiss Cheese

With enough layers, even Swiss cheese can stop most failures

- All the holes have to line up exactly for a failure to occur


Hazards
Holes are due to active failures or latent conditions
Successive layers of defenses

Source: Quality and Safety of Minimally Invasive Surgery: Past, Present, and Future

Fault-Resistant Systems (ctd)

Need to constrain control and data flow in such a manner that error propagation through the entire system is (highly) unlikely

- Or at least to minimise the occurrence of faults as much as possible

Turn the Swiss Cheese Model (of Fault Prevention) into programming practice

- Enough layers of constraints ensure that faults moving processing outside the permitted envelope is unlikely

Design by Contract

Concept introduced by Bertrand Meyer in the 1980s

Basic form is that a routine must assert pre-conditions that hold before it executes and postconditions that hold after it executes

- Well-supported in languages like Eiffel (also by Meyer)

Easy to implement in C as macros

- `REQUIRES( precondition );`
- `ENSURES( postcondition );`

```
#define REQUIRES( x )   if( !( x ) ) throw_error();
```

Design by Contract (ctd)

```
int getItem( const CRYPT_KEYID_TYPE keyIDtype,
            const void *keyID, const int keyIDlength,
            const KEYMGMT_ITEM_TYPE itemType,
            const int options )
      {
      REQUIRES( keyIDtype == CRYPT_KEYID_NAME || \
               keyIDtype == CRYPT_IKEYID_KEYID );
      REQUIRES( keyIDlength >= MIN_NAME_LENGTH && \
               keyIDlength < MAX_ATTRIBUTE_SIZE );
      REQUIRES( itemType == KEYMGMT_ITEM_PUBLICKEY );
      REQUIRES( isFlagRangeZ( options, KEYMGMT ) );
      ...
```

Design by Contract (ctd)

Function’s design-by-contract precondition is that input parameters are as expected

- Basic good sanitation
- Catches wild jumps into code
  - Precondition won’t be met
  - Using ROP to jump into exportPrivateKey() will be caught

Fundamental building block for most later safety measures

Range Checking

Commonly-used memory copy/move/append

```
memcpy( destination, source, count );
```

The function is actually

```
void *memcpy( void *destination,
              const void *source,
              size_t num );
```

int count → size_t num means negative count value becomes a huge positive value

- Has led to a number of security vulnerabilities

Making everything unsigned is a kludge

- Bites you in locations where you actually need a signed value

Range Checking (ctd)

Create integer bounds-checking functions for the most common cases

```
• isIntegerRange( value ) →
             range( 0 ... some_bound )
• isIntegerRangeNZ( value ) →
             range( 1 ... some_bound )
• isShortIntegerRange( value ) →
             range( 0 ... some_small_bound )
• isShortIntegerRangeNZ( value ) →
             range( 1 ... some_small_bound )
```

Each memory operation can now be checked

```
REQUIRES( isShortIntegerRangeNZ( count ) );
memcpy( dest, src, count );
```

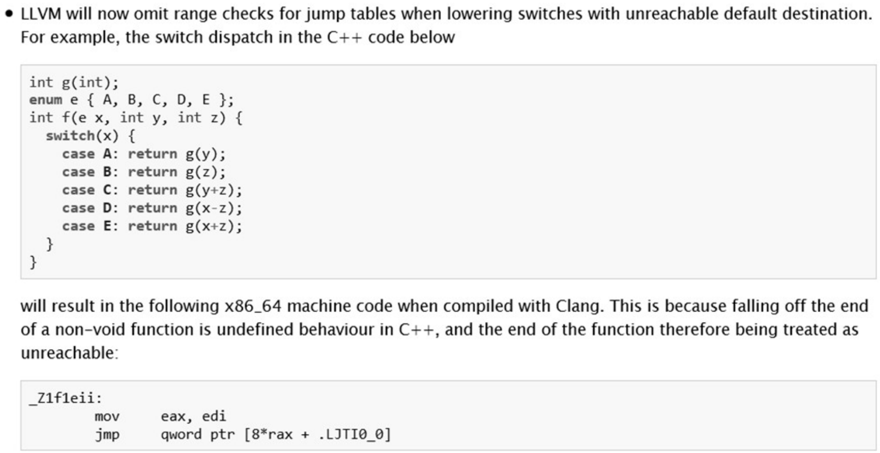
Range Checking (ctd)

Can do the same for enums and flags by following a standard naming convention when declaring them

```
typedef enum { OPERATION_NONE, OPERATION_READ,
      OPERATION_WRITE, OPERATION_EXECUTE,
      OPERATION_FORMAT, OPERATION_LAST }
      OPERATION_TYPE;

#define isEnumRange( value, name ) \
      ( value > name##_NONE && value < name##_LAST )

REQUIRES( isEnumRange( enumValue, OPERATION ) );
REQUIRES( isFlagRange( flagValue, FLAG ) );
```

Range Checking (ctd)

Can also be used to deal with compiler braindamage



- LLVM will now omit range checks for jump tables when lowering switches with unreachable default destination. For example, the switch dispatch in the C++ code below

```
int g(int);
enum e { A, B, C, D, E };
int f(e x, int y, int z) {
  switch(x) {
    case A: return g(y);
    case B: return g(z);
    case C: return g(y+z);
    case D: return g(x-z);
    case E: return g(x+z);
  }
}
```

will result in the following x86_64 machine code when compiled with Clang. This is because falling off the end of a non-void function is undefined behaviour in C++, and the end of the function therefore being treated as unreachable:

```
_Z1f1eii:
        mov     eax, edi
        jmp     qword ptr [8*rax + .LJTI0_0]
```

Bounds Checking

C has no bounds checking

- A long-standing complaint

To some extent this is turning C into Pascal/Ada/…

Need to check an index into a block of memory

- Is ‘index’ within the range { start, end } is straightforward

What about ‘is { start, length } within { 0, totalLength }’?

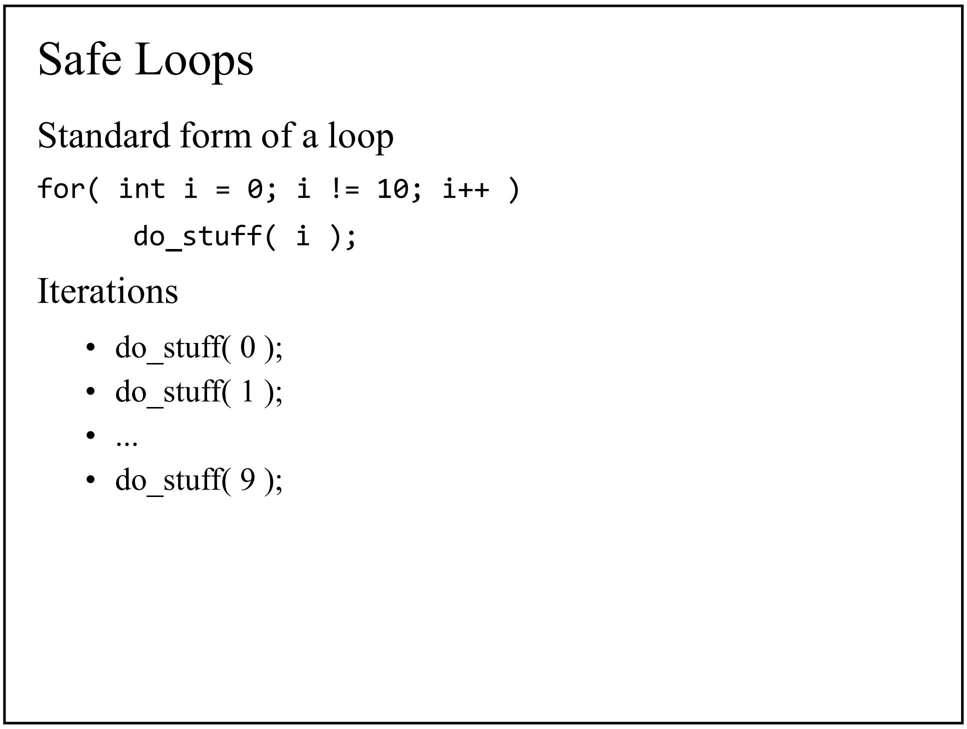
- Very common requirement when working with blocks of memory
- Also very common exploit vector, see ‘buffer overrun’

Bounds Checking (ctd)

```
#define boundsCheck( start, length, totalLength ) \
      ( ( start <= 0 || length < 1 || \
          start + length > totalLength ) ? \
      FALSE : TRUE )
```

SSH packet-assembly code:

```
REQUIRES( boundsCheck( keyDataHeaderSize,
          keyexInfoLength, receiveBufferSize ) );
memmove( keyexInfoPtr + keyDataHeaderSize,
         keyexInfoPtr, keyexInfoLength );
```



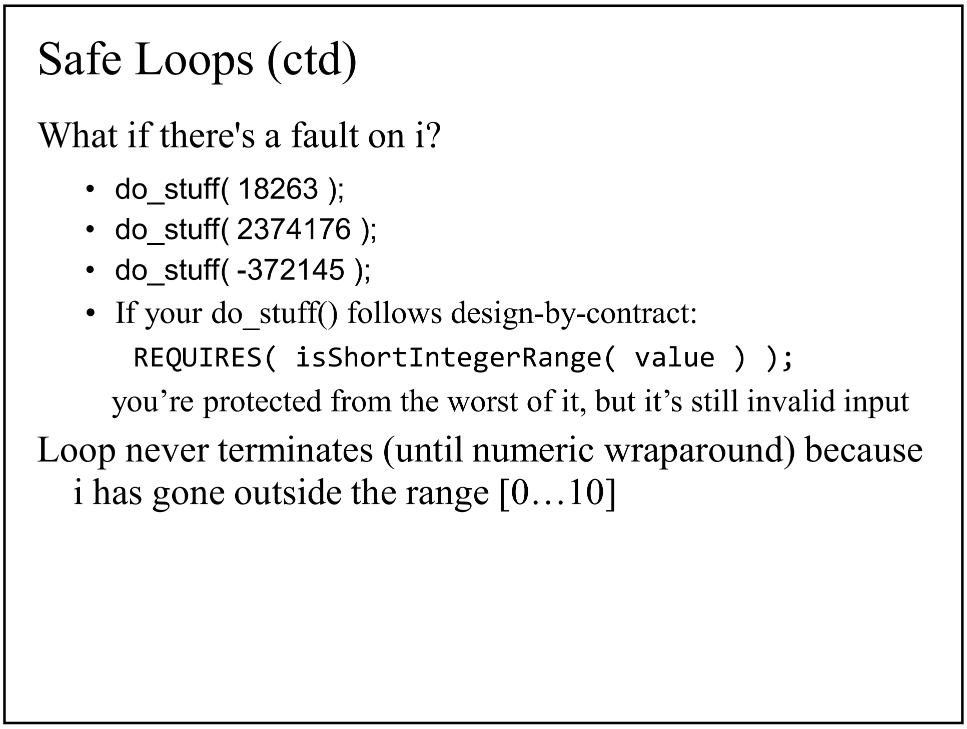
Safe Loops

Standard form of a loop

```
for( int i = 0; i != 10; i++ )
      do_stuff( i );
```

Iterations

- do_stuff(0);
- do_stuff(1);
- ...
- do_stuff(9);



Safe Loops (ctd)

What if there's a fault on i?

- do_stuff(18263);
- do_stuff(2374176);
- do_stuff(-372145);
- If your do_stuff() follows design-by-contract:

  ```
  REQUIRES( isShortIntegerRange( value ) );
  ```

  you’re protected from the worst of it, but it’s still invalid input

Loop never terminates (until numeric wraparound) because i has gone outside the range [0…10]

Safe Loops, Attempt #1

Make loop variables unsigned, use less-than rather than equality comparison

```
for( unsigned int i = 0; i < 10; i++ )
      do_stuff( i );
```

Will this loop terminate?

- Yes, for a simple loop
- Not necessarily, for a complex loop

```
      for( unsigned int i = 0; i < 10; i++ )
            i = complex_calculation();
```

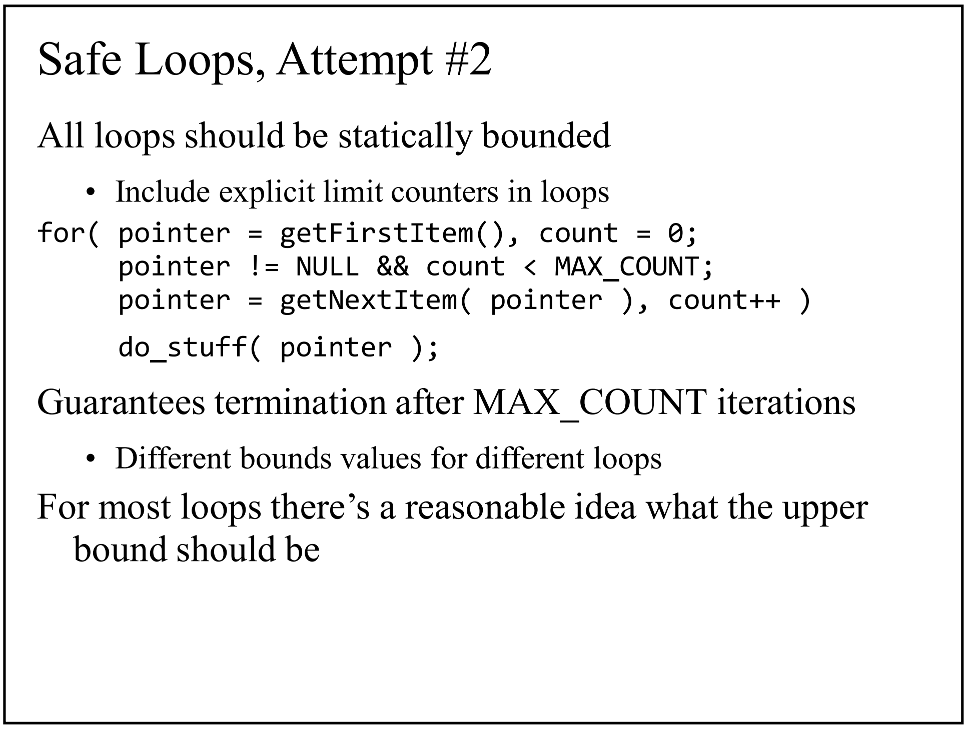
Actually even the simple loop may not work, see later slides

Safe Loops, Attempt #1 (ctd)

Will this loop terminate?

```
thing_t *pointer;
for( pointer = getFirstItem(); pointer != NULL; \
     pointer = getNextItem( pointer ) )
     do_stuff( pointer );
```

Well, it's *supposed* to terminate…



Safe Loops, Attempt #2

All loops should be statically bounded

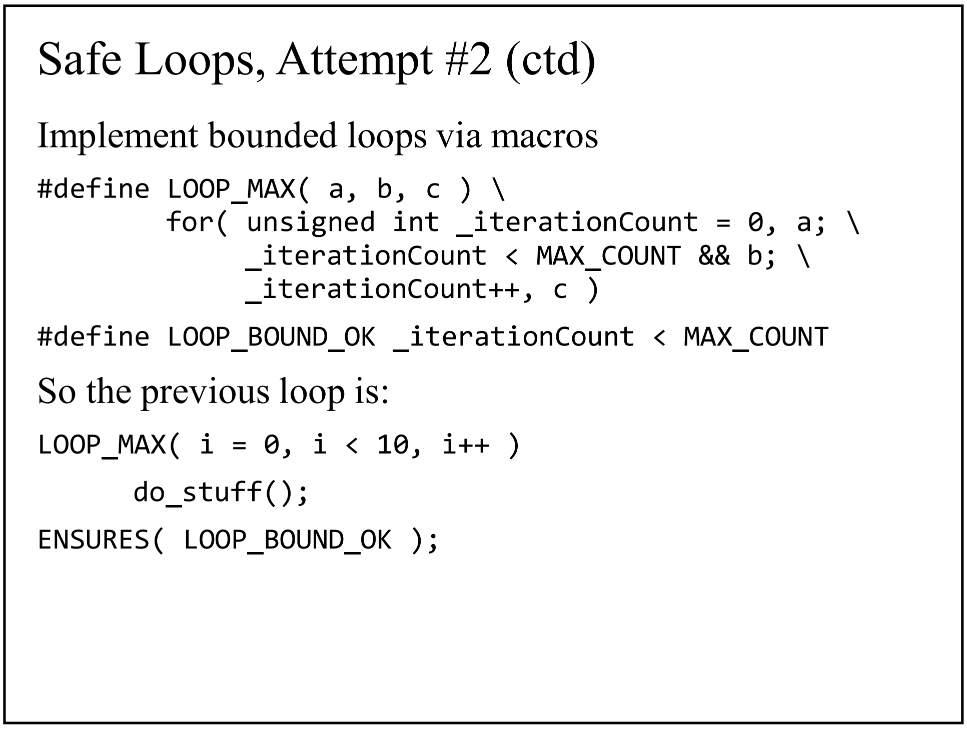
- Include explicit limit counters in loops

```
for( pointer = getFirstItem(), count = 0;
     pointer != NULL && count < MAX_COUNT;
     pointer = getNextItem( pointer ), count++ )

     do_stuff( pointer );
```

Guarantees termination after MAX_COUNT iterations

- Different bounds values for different loops

For most loops there’s a reasonable idea what the upper bound should be



Safe Loops, Attempt #2 (ctd)

Implement bounded loops via macros

```
#define LOOP_MAX( a, b, c ) \
        for( unsigned int _iterationCount = 0, a; \
             _iterationCount < MAX_COUNT && b; \
             _iterationCount++, c )

#define LOOP_BOUND_OK _iterationCount < MAX_COUNT
```

So the previous loop is:

```
LOOP_MAX( i = 0, i < 10, i++ )

      do_stuff();

ENSURES( LOOP_BOUND_OK );
```

Safe Loops, Attempt #2 (ctd)

And then the compiler screws it up

```
for( unsigned int _iterationCount = 0, i = 0; \
     _iterationCount < MAX_COUNT && i < 10; \
     _iterationCount++, i++ )
     do_stuff();
```

Merge the two loops, since both are incrementing the same value and 10 < MAX_COUNT

```
for( __x = 0; __x < 10; __x++ )
      do_stuff();
```

Safe Loops, Attempt #3

Need to have one variable counting up and the other down:

```
#define LOOP_MAX( a, b, c ) \
        for( signed int _iterationCount = \
                              MAX_COUNT, a; \
             _iterationCount > 0 && b; \
             _iterationCount--, c )
#define LOOP_BOUND_OK _iterationCount > 0
```

Loops that count down have the second variable counting up instead:

```
#define LOOP_MAX( a, b, c ) \
        for( signed int _iterationCount = 0, a; \
             _iterationCount < MAX_COUNT && b; \
             _iterationCount++, c )
#define LOOP_BOUND_OK _iterationCount < MAX_COUNT
```

Safe Loops, Attempt #3 (ctd)

The expanded form is then

```
for( signed int _iterationCount = MAX_COUNT, i = 0;
      _iterationCount > 0 && i < 10;
      _iterationCount--, i++ )
      do_stuff();
ENSURES( _iterationCount > 0 );
```

Now all loops are statically bounded and we can guarantee termination

Safe Loops, Attempt #3 (ctd)

And then the compiler screws it up again…

Wait, what could possibly go wrong with:

```
      int array[ 512 ];
      int i;
      unsigned int j;
      for( i = 0; i < 512; i++ )
            array[ i ] = 5;
      for( j = 0; j < 512; j++ )
            array[ j ] = 3;
```

gcc –O1 –S loops.c / gcc –O2 –S loops.c

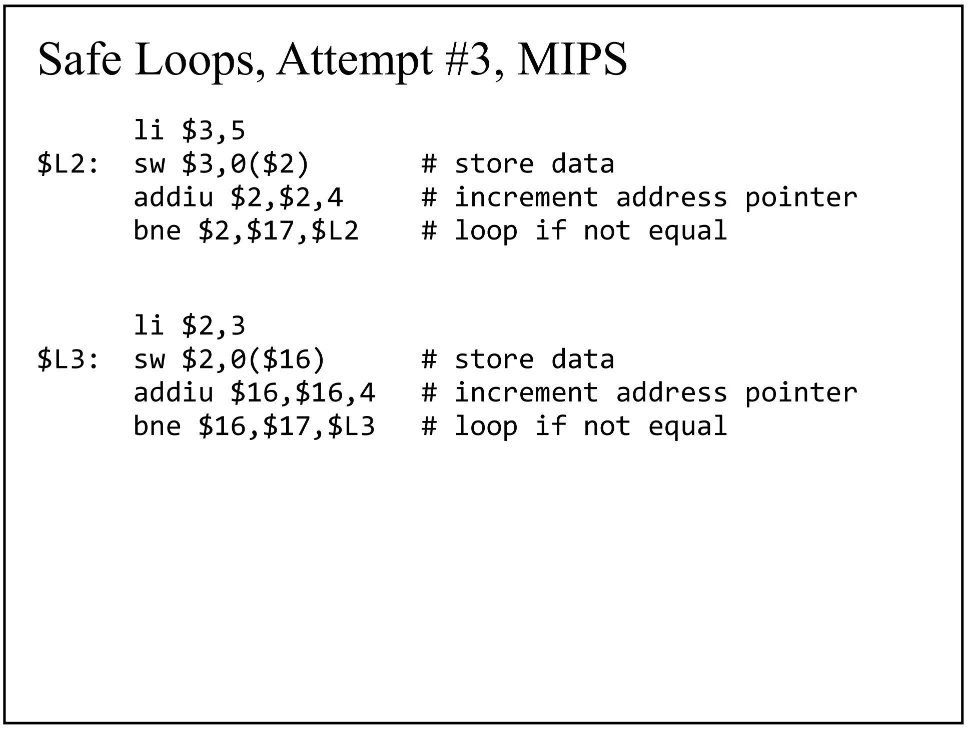
- -O3 is similar but vectorised

Safe Loops, Attempt #3, x86

```
.L2:   movl $5, (%rax)     # store $5 to address
       addq $4, %rax       # increment address pointer
       cmpq %rbp, %rax     # compare to bound
       jne .L2             # loop if not equal

.L3:   movl $3, (%rbx)     # store $3 to address
       addq $4, %rbx       # increment address pointer
       cmpq %rbp, %rbx     # compare to bound
       jne .L3             # loop if not equal
```

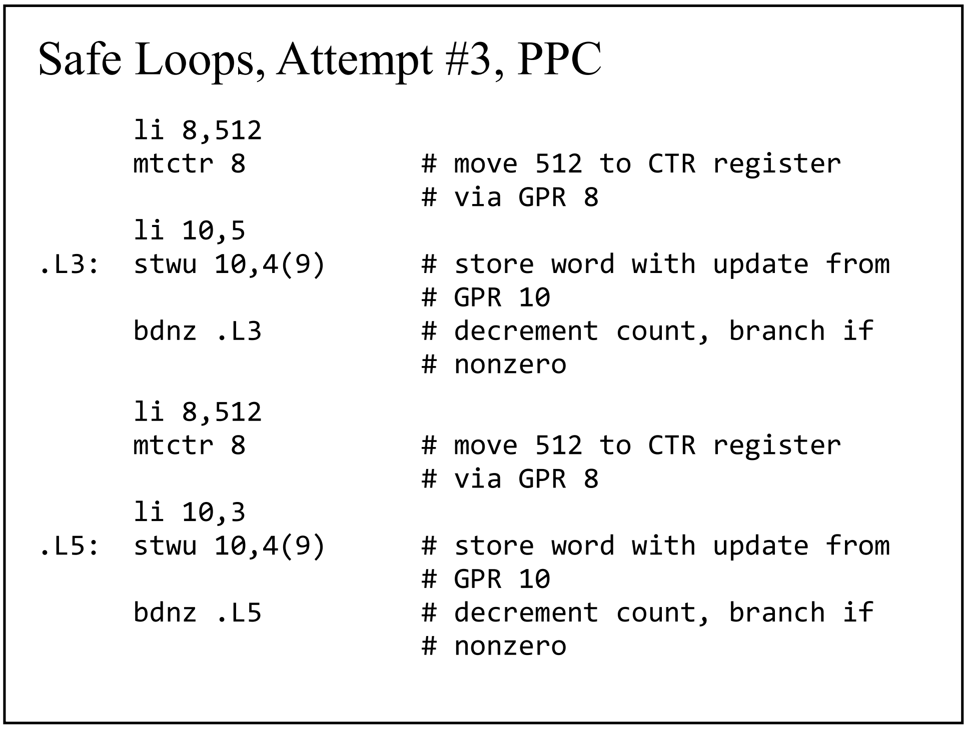
Safe Loops, Attempt #3, Arm

```
       mov w1, 5
.L3:   str w1, [x0],4      # store word from W1 to address
                           # X0, add 4
       cmp x0, x19         # compare to limit in X19
       bne .L3             # loop if not equal

       mov w1, 3
.L5:   str w1, [x0],4      # store word from W1 to address
                           # X0, add 4
       cmp x0, x19         # compare to limit in X19
       bne .L5             # loop if not equal
```



Safe Loops, Attempt #3, MIPS

```
      li $3,5
$L2:  sw $3,0($2)       # store data
      addiu $2,$2,4     # increment address pointer
      bne $2,$17,$L2    # loop if not equal


      li $2,3
$L3:  sw $2,0($16)      # store data
      addiu $16,$16,4   # increment address pointer
      bne $16,$17,$L3   # loop if not equal
```



Safe Loops, Attempt #3, PPC

```
      li 8,512
      mtctr 8           # move 512 to CTR register
                        # via GPR 8
      li 10,5
.L3:  stwu 10,4(9)      # store word with update from
                        # GPR 10
      bdnz .L3          # decrement count, branch if
                        # nonzero
      li 8,512
      mtctr 8           # move 512 to CTR register
                        # via GPR 8
      li 10,3
.L5:  stwu 10,4(9)      # store word with update from
                        # GPR 10
      bdnz .L5          # decrement count, branch if
                        # nonzero
```

Safe Loops, Attempt #3, RISC-V

```
      li a4,5
.L2:  sw a4,0(a5)       # store word in A4 in address
      addi a5,a5,4      # increment address pointer
      bne a5,s1,.L2     # branch if address less than
                        # bound


      li a5,3
.L3:  sw a5,0(s0)       # store word in A4 in address
      addi s0,s0,4      # increment address pointer
      bne s0,s1,.L3     # branch if address less than
                        # bound
```

Safe Loops, Attempt #3, Sparc

```
      mov 5, %g2
      st %g2, [%g1]
.L7:  add %g1, 4, %g1   # add 4
      cmp %g1, %i4      # compare to bound
      bne %xcc, .L7     # loop if not equal
      st %g2, [%g1]     # store data in delay slot
      mov 3, %g1
      st %g1, [%i5]
.L8:  add %i5, 4, %i5   # add 4
      cmp %i5, %i4      # compare to bound
      bne %xcc, .L8     # loop if not equal
      st %g1, [%i5]     # store data in delay slot
```

Safe Loops, Attempt #3 (ctd)

This isn’t architecture-specific

- It’s universal across all gcc-produced code

gcc converts all loops from a safe [0…n] index bound to an unsafe index != n

- No (known) version of gcc will compile this loop correctly
- “Correctly” = preserving the semantics of the original code

Safe Loops, Attempt #3 (ctd)

gcc bonus feature: If there’s a bounds check within the loop…

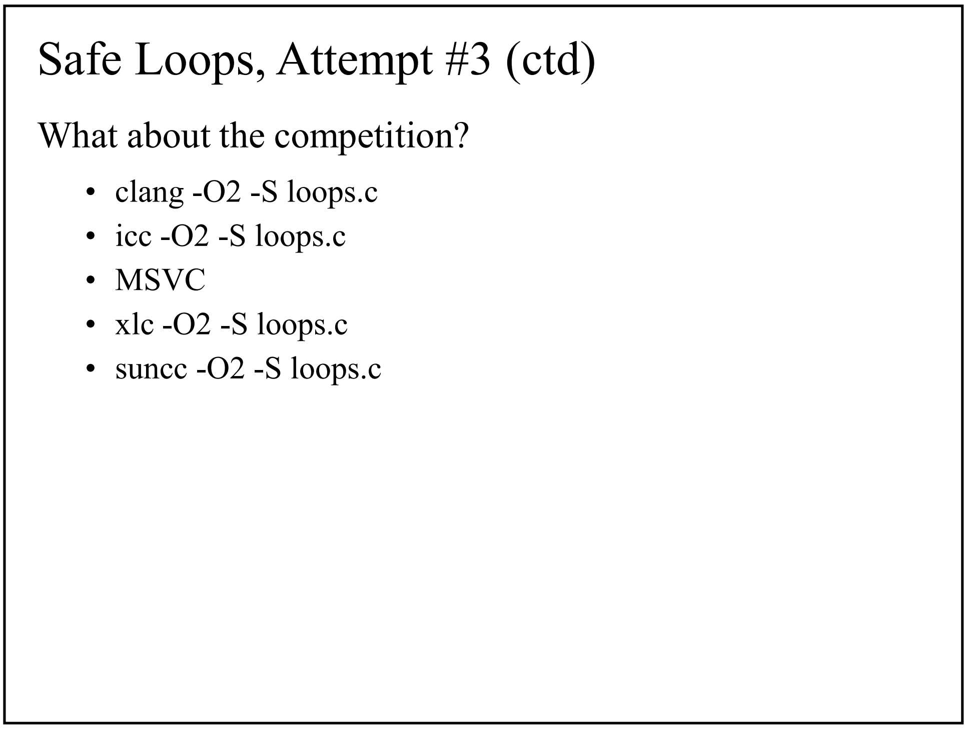
```
XXX: note: in expansion of macro 'boundsCheck’
XXX: warning: comparison of unsigned expression
     >= 0 is always true [-Wtype-limits]
```

Emitted code treats loop index as signed

- Or at least don’t-care, via = / !=

Emitted code treats bounds check as unsigned and removes it

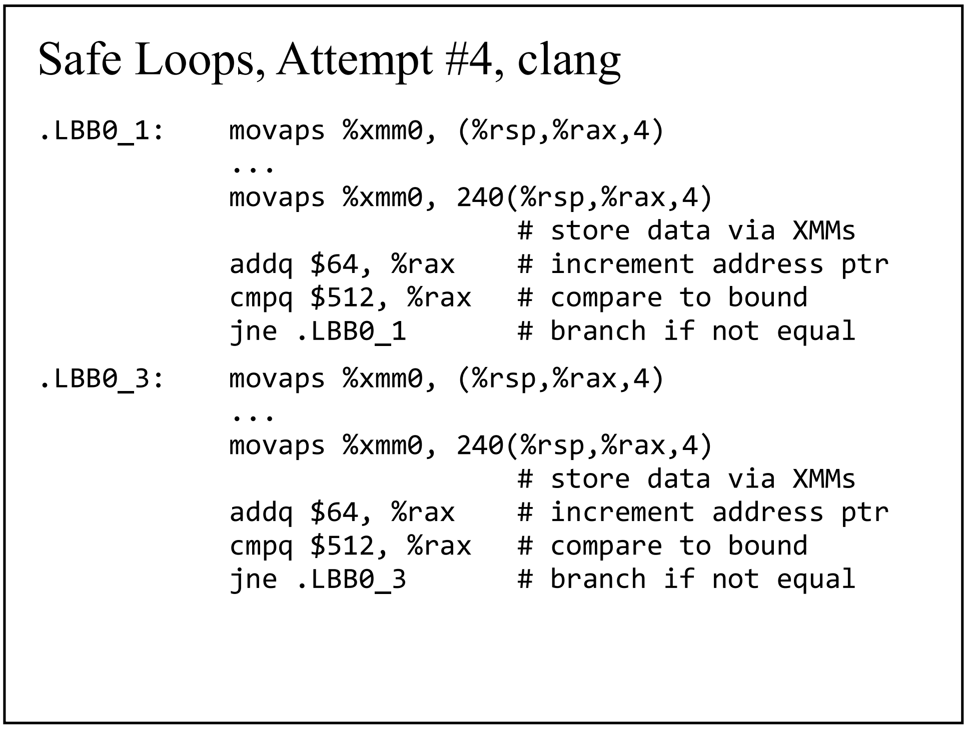
- Loses both safe-loop and safe-bounds operations in one go



Safe Loops, Attempt #3 (ctd)

What about the competition?

- clang -O2 -S loops.c
- icc -O2 -S loops.c
- MSVC
- xlc -O2 -S loops.c
- suncc -O2 -S loops.c



Safe Loops, Attempt #4, clang

```
.LBB0_1:    movaps %xmm0, (%rsp,%rax,4)
            ...
            movaps %xmm0, 240(%rsp,%rax,4)
                              # store data via XMMs
            addq $64, %rax    # increment address ptr
            cmpq $512, %rax   # compare to bound
            jne .LBB0_1       # branch if not equal
.LBB0_3:    movaps %xmm0, (%rsp,%rax,4)
            ...
            movaps %xmm0, 240(%rsp,%rax,4)
                              # store data via XMMs
            addq $64, %rax    # increment address ptr
            cmpq $512, %rax   # compare to bound
            jne .LBB0_3       # branch if not equal
```

Safe Loops, Attempt #4, clang (ctd)

```
            movi v0.4s, #5    # vector load data
.LBB0_1:    add x10, x9, x8
            add x8, x8, #32   # increment address ptr
            cmp x8, #2048     # compare to bound
            stp q0, q0, [x10] # store quadword reg pair
            b.ne .LBB0_1      # branch if not equal

            movi v0.4s, #3    # vector load data
.LBB0_3:    add x9, x19, x8
            add x8, x8, #32   # increment address ptr
            cmp x8, #2048     # compare to bound
            stp q0, q0, [x9]  # store quadword reg pair
            b.ne .LBB0_3      # branch if not equal
```

Safe Loops, Attempt #4, icc

```
..B1.2:     movdqu XMMWORD PTR [rsp+rax*4], xmm0
            ...
            movdqu XMMWORD PTR [48+rsp+rax*4], xmm0
            add rax, 16       # increment address ptr
            cmp rax, 512      # compare to bound
            jb ..B1.2         # branch if below

..B1.5:     lea edx, DWORD PTR [4+rax]
            movdqu XMMWORD PTR [rsp+rdx*4], xmm0
            ...
            lea esi, DWORD PTR [12+rax]
            movdqu XMMWORD PTR [rsp+rax*4], xmm0
            add eax, 16       # increment address ptr
            cmp eax, 512      # compare to bound
            jb ..B1.5         # branch if below
```

Unsigned
comparison/
branch!

Safe Loops, Attempt #4, MSVC

```
           mov ecx, 32
           mov rdx, 0000000500000005H
$LL1@main:mov QWORD PTR [rax], rdx
           ...
           mov QWORD PTR [rax+16], rdx
           lea rax, QWORD PTR [rax+64]
           mov QWORD PTR [rax-40], rdx
           ...
           mov QWORD PTR [rax-8], rdx    # store data
           sub rcx, 1                    # dec count
           jne SHORT $LL1@main           # branch if
                                         # not equal

           […]
```

Safe Loops, Attempt #4, MSVC (ctd)

```
           […]

           mov ebx, 32
           mov rcx, 0000000300000003H
$LL2@main:mov QWORD PTR [rax], rcx
           ...
           mov QWORD PTR [rax+16], rcx
           lea rax, QWORD PTR [rax+64]
           mov QWORD PTR [rax-40], rcx
           ...
           mov QWORD PTR [rax-8], rcx    # store data
           sub rbx, 1                    # dec count
           jne SHORT $LL2@main           # branch if
                                         # not equal
```

Safe Loops, Attempt #4, xlc

```
      cal r4,64(r0)
      mtspr CTR,r4      # move 64 to CTR reg via GPR 4
__L30:st r0,4(r3)
      ...
      st r0,32(r3)      # store data, unrolled
      cal r3,32(r3)     # add 32 to address
      bc BO_dCTR_NZERO,CR0_LT,__L30
                        # branch if counter nonzero
      cal r4,64(r0)
      mtspr CTR,r4      # move 64 to CTR reg via GPR 4
__L80:st r0,4(r31)
      ...
      st r0,32(r31)     # store data, unrolled
      cal r31,32(r31)   # add 32 to address
      bc BO_dCTR_NZERO,CR0_LT,__L80
                        # branch if counter nonzero
```

Safe Loops, Attempt #4, suncc

Correct signed/
unsigned branch!

```
      or %g0,5,%i5
      st %i5,[%i1]
      or %g0,0,%i0      # zero counter
.L19: add %i0,1,%i0     # increment counter
      add %i1,4,%i1     # increment address
      cmp %i0,511       # compare to bound
      ble %icc,.L19     # branch if less or equal
      st %i5,[%i1]      # store data in delay slot
      or %g0,3,%i4
      sll %i2,2,%i5
.L18: add %i2,1,%i2     # increment counter
      st %i4,[%i5+%i3]  # store data
      cmp %i2,511       # compare to bound
      bleu %icc,.L18    # branch if less or equal,
                        # unsigned
      sll %i2,2,%i5     # address = counter * 4 in
                        # delay slot
```

Safe Loops, Attempt #4 (ctd)

There exists at least one compiler, running on at least one computer, which will compile the safe-loop code correctly

Need to defeat the compiler’s ~~braindamage~~ optimiser

- Add invariant check in loop body

```
ENSURES( LOOP_INVARIANT( i, 0, 10 ) );
```

- See later slides

Safe Loops, Attempt #4 (ctd)

Examples from the real-time control world

- Compile with optimisation disabled since this destroys the 1:1 mapping of source → object code
  - IEC 61508-3 §7.4.4.4 / ISO 26262-8 §11.4.4.2 warn against optimising compilers
- Build on 1990s-vintage PCs scrounged from eBay because that’s what was certified
- See “Automotive Control Systems Security” talk

Source: eBay

Safe Loops, Attempt #4 (ctd)

Does disabling optimisation really fix things?

gcc –O0 -S loops.c

```
      jmp .L2
.L3:  movl -4(%rbp), %eax     # load address
      cltq                    # convert long to quad
      movl $5, -2064(%rbp,%rax,4) # store data
      addl $1, -4(%rbp)       # increment address
.L2:  cmpl $511, -4(%rbp)     # compare to bound
      jle .L3                 # branch if less or equal
```

Correct signed branch

Safe Loops, Attempt #4 (ctd)

```
      jmp .L4
.L5:  movl -8(%rbp), %eax     # load address
      movl $3, -2064(%rbp,%rax,4) # store data
      addl $1, -8(%rbp)       # increment address
.L4:  cmpl $511, -8(%rbp)     # compare to bound
      jbe .L5                 # branch if below or eq
```

Correct unsigned
branch

- Yes, but the resulting code is pretty bad

Safe Loops, Attempt #4 (ctd)

What about CompCert?

- Formally verified optimizing compiler developed at INRIA, France

  > “Mathematical proof that the generated executable code behaves exactly as prescribed by the semantics of the source program”
  > — CompCert documentation

- Mechanism for getting people to swear in French

Safe Loops, Attempt #5

ccomp –O2 –S loops.c

Correct signed/
unsigned branch

```
.L10: leaq 8(%rsp), %rcx
      movslq %r9d, %r10
      movl $5, %r8d
      movl %r8d, 0(%rcx,%r10,4) # store data
      leal 1(%r9d), %r9d
      cmpl $512, %r9d   # compare to bound
      jl .L10           # branch if less than
.L11: leaq 8(%rsp), %rax
      movl %edx, %edi
      movl $3, %esi
      movl %esi, 0(%rax,%rdi,4) # store data
      leal 1(%edx), %edx
      cmpl $512, %edx   # compare to bound
      jb .L11           # branch if less than, unsigned
```

Safe Loops, Attempt #5 (ctd)

Correct as advertised, but not the most optimal of code

- Lots of unnecessary memory loads and register transfers
- About as bad as gcc –O0

Is this a side-effect of semantics-preserving transformations, or just poor code generation?

Loop Invariants

We know that we got to the end of the loop OK, but what happens inside the loop body?

- If a fault happens while executing the loop, the postcondition is met but the loop wasn’t executed as intended

```
for( signed int _iterationCount = MAX_COUNT, i = 0;
     _iterationCount > 0 && i < 10;
     _iterationCount--, i++ )
     do_stuff();
ENSURES( _iterationCount > 0 );
```

- Exit at i = 7, _iterationCount > 0 so all appears OK

Loop Invariants (ctd)

Great for glitch attacks

- Glitch a password-checking loop to bypass password checks

Timing-neutral password check loop

```
      ld r0, 0
      ld r1, 16
loop: ld r2, requiredPassword[ i ]
      xor r2, userPassword[ i ]
      or r0, r2
      dec r1                 Glitch
      jnz loop
```

Clock glitch steps the PC twice but the ALU only once

- Break out of the loop after checking only one character of the password

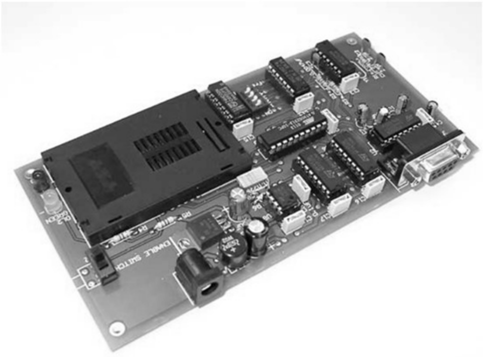
Loop Invariants (ctd)

Long history of attacks going back to the 1990s with smart card unloopers

- DOS-based control software run over serial port

Today more likely to be voltage or EM glitches

- Inject EMI via probe
- Voltage brownout
- Reset-signal glitch



Source: Cellularenter

Loop Invariants (ctd)

Address by using loop invariants

```
for( signed int _iterationCount = MAX_COUNT, i = 0;
     _iterationCount > 0 && i < 10;
     _iterationCount--, i++ )
     do_stuff();
ENSURES( _iterationCount > 0 );
```

Note that the ratio between the two loop counters remains constant

- i + _iterationCount == MAX_COUNT at all times

Loop Invariants (ctd)

Add a loop invariant check

```
#define LOOP_INVARIANT( index, lowerBound, \
                        upperBound ) \
        ( index >= lowerBound && \
          index <= upperBound && \
          ( index - lowerBound + \
                  _iterationCount == MAX_COUNT ) )
```

- With the extras for reverse loops and ones where there's no fixed relationship between the two loop variables

Loop Invariants (ctd)

So our long-suffering loop becomes:

```
LOOP_MAX( i = 0, i < 10, i++ )
      LOOP_INVARIANT( i, 0, 10 );
      do_stuff();
ENSURES( LOOP_BOUND_OK );
```

- The expanded macro form is pretty ugly, not shown here

For fixed-iteration loops, also check that i == 10 at the end

Array Bounds

Static arrays have fixed bounds

- Can be evaluated at compile time

```
      #define ARRAYSIZE( array, elementType ) \
            ( ( sizeof( array ) / \
                sizeof( elementType ) ) - 1 )
```

Overallocate all (static) arrays by one element

```
thing_t array[] = { thing1, thing2, thing3, thing4,
                    NULL, NULL };
for( unsigned int i = 0; array[ i ] != NULL; i++ )
      do_stuff( array[ i ];
```

Array Bounds (ctd)

Safe version is:

```
for( signed int _iterationCount = \
            ARRAYSIZE( array, thing_t ), i = 0;
     _iterationCount > 0 && array[ i ] != NULL;
     _iterationCount--, i++ )
     doStuff( array[ i ] );
```

If the soft bound of array[i] != NULL isn't hit then the hard array-size bound is triggered

Safe Pointers

```
thing_t *pointer;
for( pointer = getFirstItem(); pointer != NULL; \
     pointer = getNextItem( pointer ) )
     do_stuff( pointer );
```

Let's make the loop safe

```
thing_t * pointer;
LOOP_MAX( pointer = getFirstItem(), \
          pointer != NULL, \
          pointer = getNextItem( pointer ) )
      do_stuff( pointer );
```

This will terminate, but we don't know where the pointers will end up going before the hard bound is triggered

Safe Pointers (ctd)

Pointers are two-valued

- NULL = invalid/not set
- Anything else = (apparently) valid

Should be tri-state

- NULL
- Valid pointer to item
- Invalid pointer

Safe Pointers (ctd)

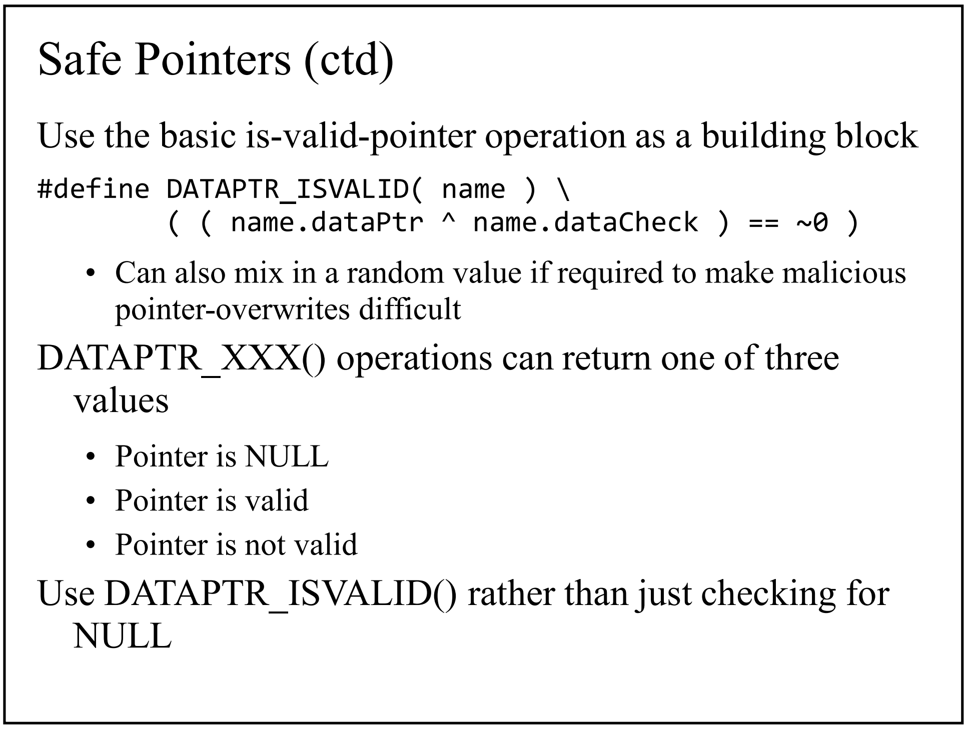
Turn pointers from vectors into scalars

- Store a pointer and its complement

```
typedef struct {
      void *dataPtr;
      uintptr_t dataCheck;
      } DATAPTR;
```

Function pointers are special because of things like IA64's "totally idiotic calling conventions" (Linus)

- Hide them behind macros, not important here



Safe Pointers (ctd)

Use the basic is-valid-pointer operation as a building block

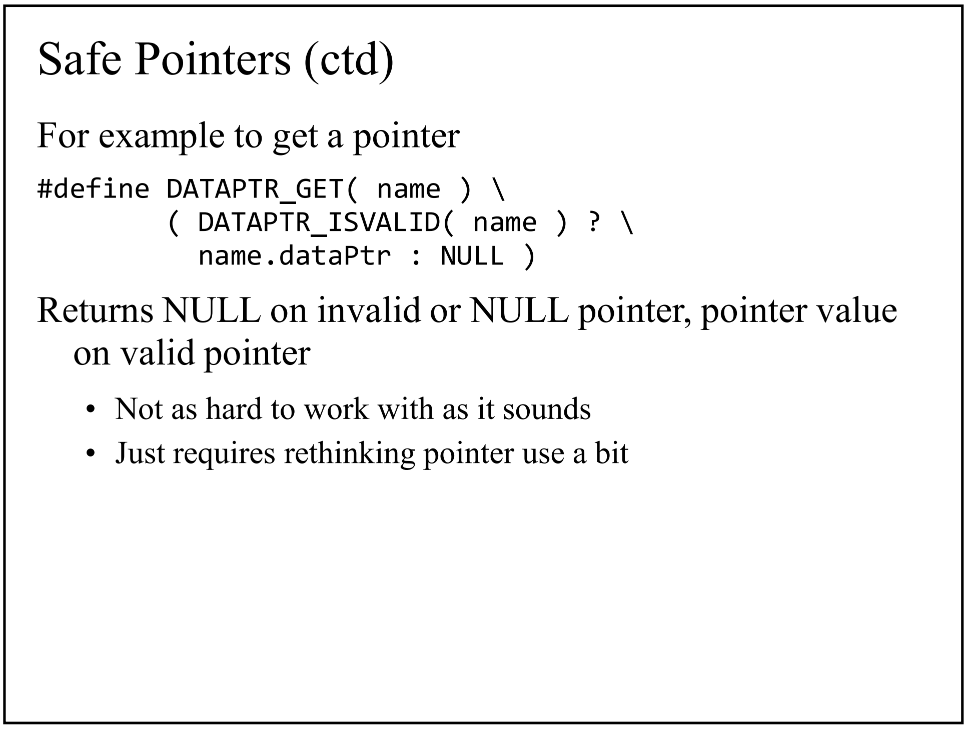
```
#define DATAPTR_ISVALID( name ) \
        ( ( name.dataPtr ^ name.dataCheck ) == ~0 )
```

- Can also mix in a random value if required to make malicious pointer-overwrites difficult

DATAPTR_XXX() operations can return one of three values

- Pointer is NULL
- Pointer is valid
- Pointer is not valid

Use DATAPTR_ISVALID() rather than just checking for NULL



Safe Pointers (ctd)

For example to get a pointer

```
#define DATAPTR_GET( name ) \
        ( DATAPTR_ISVALID( name ) ? \
          name.dataPtr : NULL )
```

Returns NULL on invalid or NULL pointer, pointer value on valid pointer

- Not as hard to work with as it sounds
- Just requires rethinking pointer use a bit

Safe Pointers (ctd)

Standard list-walking loop

```
LOOP_LARGE( listPtr = DATAPTR_GET( listHead ),
            listPtr != NULL,
            listPtr = DATAPTR_GET( listPtr->next ) )
            do_thing( listPtr );
```

Bounded loop guaranteed to pass a valid pointer to do_thing()

- Can add a DATAPTR_ISVALID() check if you need a hard error on an invalid pointer rather than just exiting the loop

Safe Booleans

```
#define FALSE 0
#define TRUE 1
```

Yes-biased boolean

- One FALSE value
- 4,294,967,295 TRUE values

Example of booleans that shouldn’t be yes-biased

- Access authorised
- Cryptographic verification succeeded
- Eject reactor core

Almost any fault or (malicious) overwrite of any kind will set a boolean to TRUE

Safe Booleans (ctd)

NXP LPC devices notoriously used one of the following values to flag security measures

- 0x12345678, 0x87654321, 0x43218765, and 0x4E697370 (‘Nisp’) = Enabled
- Remaining ~4 billion values = Disabled

STM’s config was no better

- { 0xCC, 0x33 } = High security
- { 0xAA, 0x55 } = No security
- Remaining 64K - 2 values = Medium/low security

Safe Booleans (ctd)

Should be:

- One FALSE value
- One TRUE value
- 4,294,967,294 INVALID values

> The values of each configuration datum SHALL be stored as distinctive multibit values such that no single or double bit corruption would lead to another valid value
>
> — “Embedded Software Development for Safety-Critical Systems”, p.73.

Safe Booleans (ctd)

Store bit patterns calculated to be most vulnerable to SEUs

0000	0000	1111	1111	0011	0011	1100	1100
0	0	F	F	3	3	C	C
0000	1111	0011	1100	0101	0110	1001	1111
0	F	3	C	5	6	9	F

```
#define TRUE 0x0F3C569F
```

- This is for radiation-induced upsets, for security store an unpredictable pattern

Safe Booleans (ctd)

However, the compiler strikes again

> The means of storing multibit values [...] SHALL be such that the compiler does not reduce them to a single bit, irrespective of the optimisation level used
>
> — "Embedded Software Development for Safety-Critical Systems", p.73.

Apply design-by-contract again

```
int doThing( …, BOOLEAN doOtherThing, … )
      {
      REQUIRES( doOtherThing == TRUE || \
                doOtherThing == FALSE );
      …
```

Safe Integers

Requires compiler support

clang and gcc have intrinsics

```
bool __builtin_sadd_overflow( int x, int y, int *sum );
bool __builtin_smul_overflow( int x, int y,
                              int *prod );
```

Compiles to two instructions, the arithmetic operation and a setcc

Safe Integers (ctd)

Windows has ‘portable’ intsafe operations

```
HRESULT IntAdd( INT iAugend, INT iAddend,
                INT *piResult );
HRESULT IntMult( INT iMultiplicand, INT iMultiplier,
                 INT *piResult );
```

- Can produce dozens of instructions and even function calls

Ugly and messy, needs better compiler support

- Better to perform range/bounds checks during/after operations as required

Safe Buffers

Another perpetual C problem, buffer overruns

Allocate buffers with cookies/canaries at the ends

```
#define SAFEBUFFER_SIZE( size ) \
        ( SAFEBUFFER_COOKIE_SIZE + size + \
          SAFEBUFFER_COOKIE_SIZE )
#define SAFEBUFFER_PTR( buffer ) \
        ( buffer + SAFEBUFFER_COOKIE_SIZE )
```

Allocate and access buffers using the above macros

```
BYTE safeBuffer[ SAFEBUFFER_SIZE( 1024 ) ];
safeBufferInit( SAFEBUFFER_PTR( safeBuffer ), 1024 );
readData( ioStream, safeBuffer, 1024 );
```

Safe Buffers (ctd)

```
int readData( IOSTREAM ioStream, BYTE safeBuffer,
              const int safeBufferSize )
      {
      REQUIRES( safeBufferCheck( safeBuffer, \
                                 safeBufferSize ) );
      …
      do_stuff();
      …
      ENSURES( safeBufferCheck( safeBuffer, \
                                safeBufferSize ) );
      }
```

Obviously won’t catch all overwrites, but will catch off-by-one overruns/underruns

- Can tune cookie size for how large an over/underrun you want to catch

Control-Flow Integrity Checks

Make sure function B was called from function A and nowhere else

- Make sure function B was the one that was supposed to be called
- Call to exportPrivateKey() or shutdownReactor() should not be accidental

Make sure control flows through function B in the expected manner

- Apart from the obvious error control, also makes ROP a lot harder

Control-Flow Integrity Checks (ctd)

Use Bernstein hashing to identify functions and code blocks

- Good hash function for ASCII strings
- (Very) Low probability of collisions
  - Good enough, we need something that’s OK, not perfect

Done via the preprocessor

- Really beats up the compiler

Control-Flow Integrity Checks (ctd)

Calling a function with an access token

```
shutdownReactor( MK_TOKEN( "shutdownReactor", \
                           12 ), … );
```

Called function

```
int shutdownReactor( const ACCESS_TOKEN
                                    accessToken, … )
      {
      REQUIRES( CHECK_TOKEN( "shutdownReactor", 12 ) );
      …
      }
```

Control-Flow Integrity Checks (ctd)

Can also be used to enforce control-flow integrity within functions

Creates two expressions of the control flow

- Implicitly coded into the function
- Explicitly stated at the end of the function

If the final values don’t match then there’s a problem with the control flow

Control-Flow Integrity Checks (ctd)

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
      goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
      goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, \
                              &clientRandom)) != 0)
      goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, \
                              &serverRandom)) != 0)
      goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, \
                              &signedParams)) != 0)
      goto fail;
      goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
      goto fail;
```

Control-Flow Integrity Checks (ctd)

Usage is as follows

```
CFI_CHECK_TYPE CFI_CHECK_VALUE = CFI_CHECK_INIT;
```

- Sets initial value to the file and/or function name

```
code;
CFI_CHECK_UPDATE( sequencePoint1Name );
code;
CFI_CHECK_UPDATE( sequencePoint2Name );
code;
CFI_CHECK_UPDATE( sequencePoint3Name );
```

- Updates the value as each sequence point is passed

Control-Flow Integrity Checks (ctd)

```
ENSURES( \
    CFI_CHECK_SEQUENCE_3( sequencePoint1Name,
                          sequencePoint2Name,
                          sequencePoint3Name ) );
```

- Recomputes the value and checks that it matches the running total
  - Various tricks to ensure that the compiler still evaluates each value
- For portability can’t use varargs macros, so the summing-up at the end gets a bit ugly

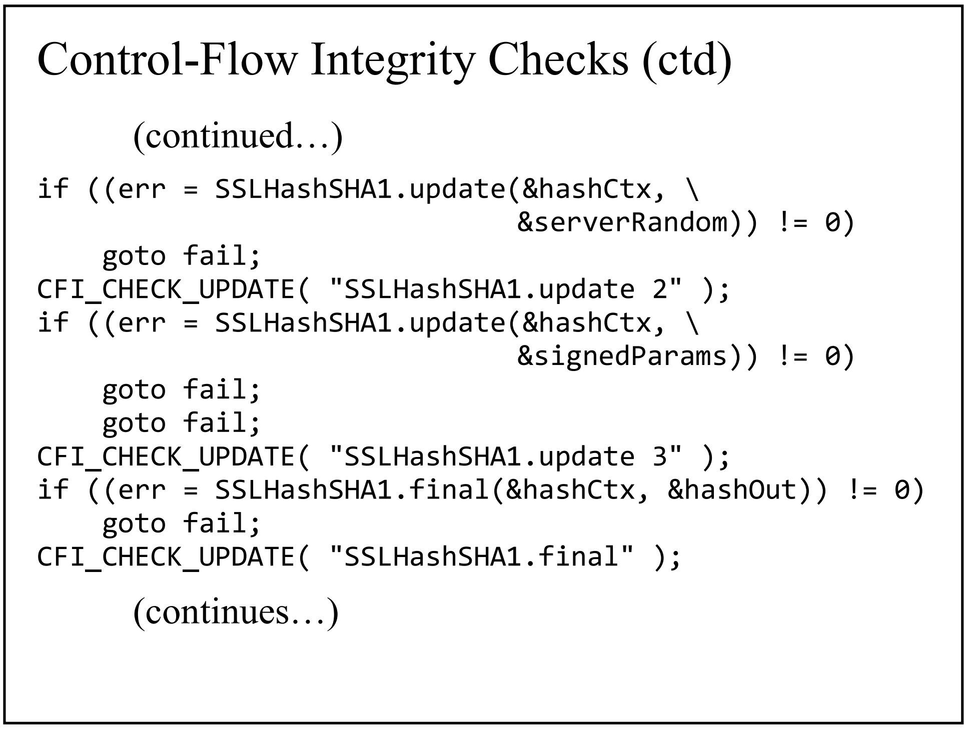
Control-Flow Integrity Checks (ctd)

Apple case would be:

```
CFI_CHECK_TYPE CFI_CHECK_VALUE = CFI_CHECK_INIT;
```

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
CFI_CHECK_UPDATE( "SSLFreeBuffer" );
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
CFI_CHECK_UPDATE( "ReadyHash" );
if ((err = SSLHashSHA1.update(&hashCtx, \
                              &clientRandom)) != 0)
    goto fail;
CFI_CHECK_UPDATE( "SSLHashSHA1.update 1" );
```

(continues…)

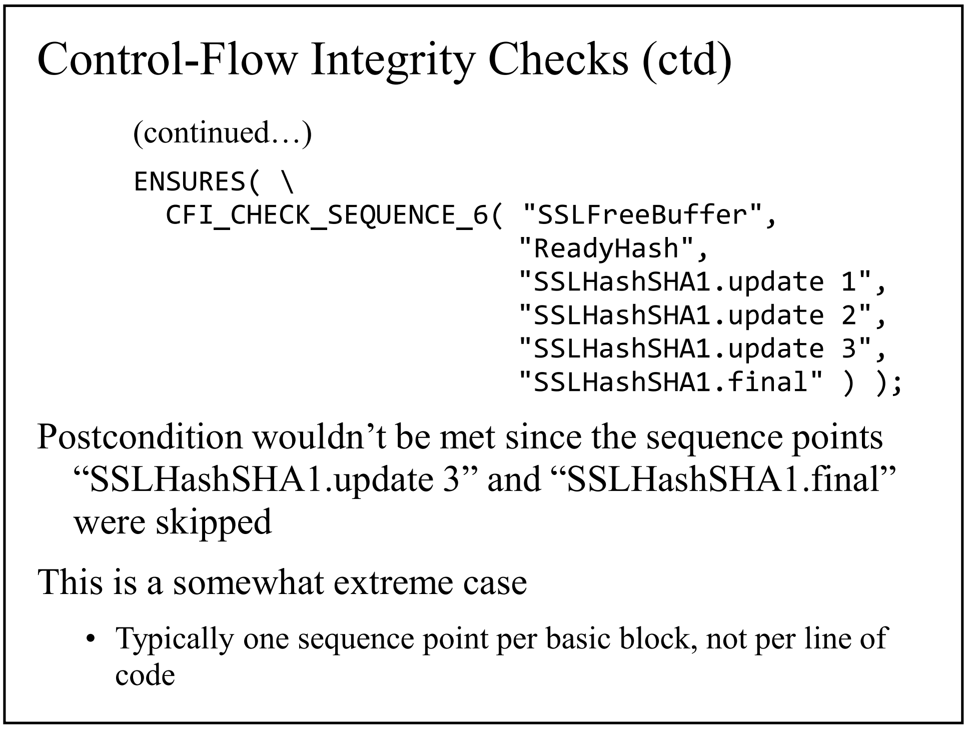


Control-Flow Integrity Checks (ctd)

(continued…)

```
if ((err = SSLHashSHA1.update(&hashCtx, \
                              &serverRandom)) != 0)
    goto fail;
CFI_CHECK_UPDATE( "SSLHashSHA1.update 2" );
if ((err = SSLHashSHA1.update(&hashCtx, \
                              &signedParams)) != 0)
    goto fail;
    goto fail;
CFI_CHECK_UPDATE( "SSLHashSHA1.update 3" );
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
CFI_CHECK_UPDATE( "SSLHashSHA1.final" );
```

(continues…)



Control-Flow Integrity Checks (ctd)

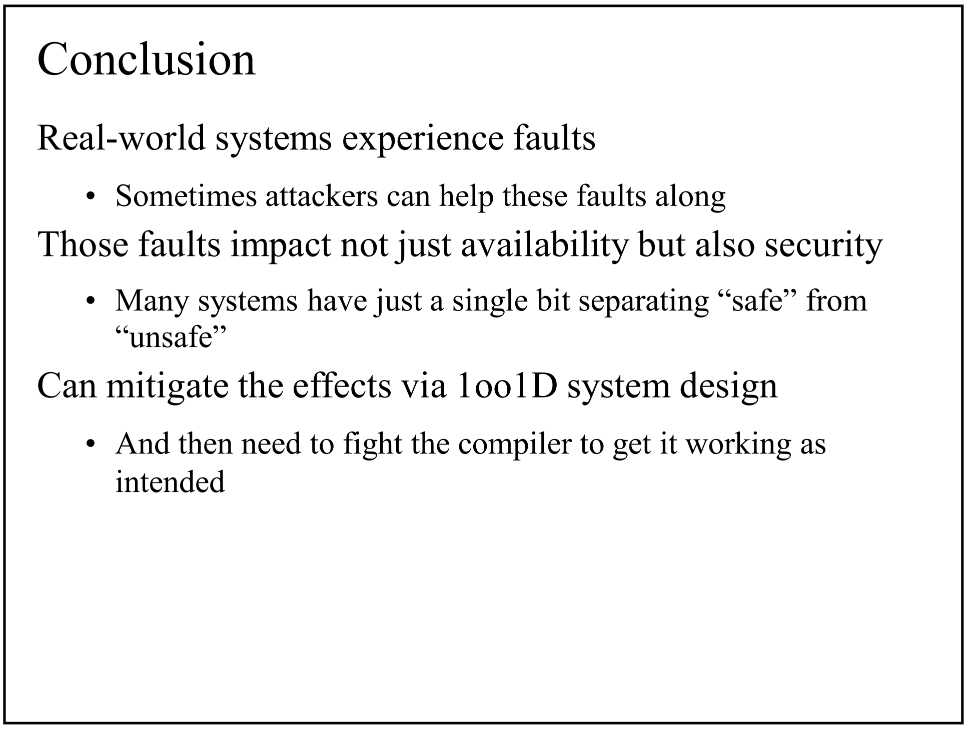
(continued…)

```
ENSURES( \
  CFI_CHECK_SEQUENCE_6( "SSLFreeBuffer",
                        "ReadyHash",
                        "SSLHashSHA1.update 1",
                        "SSLHashSHA1.update 2",
                        "SSLHashSHA1.update 3",
                        "SSLHashSHA1.final" ) );
```

Postcondition wouldn’t be met since the sequence points “SSLHashSHA1.update 3” and “SSLHashSHA1.final” were skipped

This is a somewhat extreme case

- Typically one sequence point per basic block, not per line of code



Conclusion

Real-world systems experience faults

- Sometimes attackers can help these faults along

Those faults impact not just availability but also security

- Many systems have just a single bit separating “safe” from “unsafe”

Can mitigate the effects via 1oo1D system design

- And then need to fight the compiler to get it working as intended