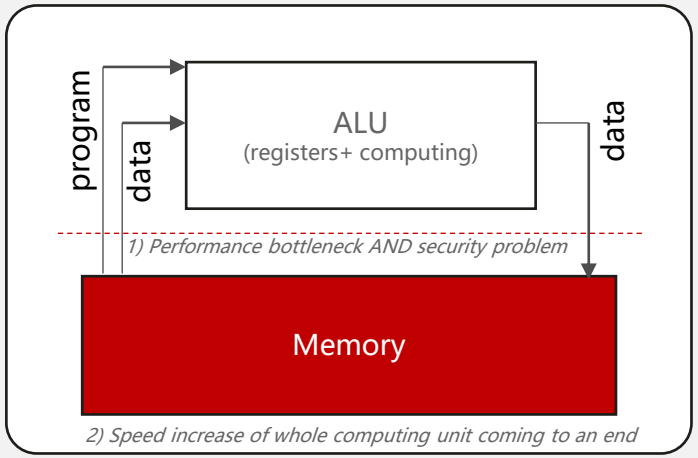
Memory Protection

1. Introduction and Motivation
2. The Compiler, and Compiler-assisted software protections
3. Hardware Mechanisms + Compiler as protection
4. Going Further. What about memory-safe languages? New Hardware?

Motivation

Von Neumann Architecture

= stored program computer



① Lack of type coherence between ALU regs/pipeline and data in memory leads to vulnerabilities
Cache & pipeline (coherence) parallellism issues in multi-core

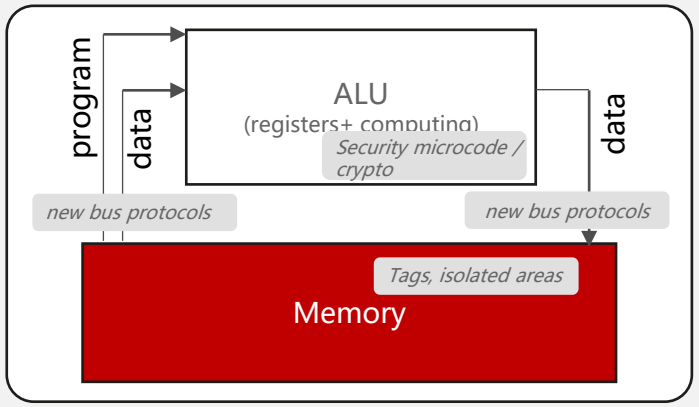
② Computing speed increase ends soon (Moore´s law)
Memory bus contention present in contemporary computing

Architectural Security (memory protection):

Add hardware features (to von Neumann) than can provide memory protection (together with compilers & SW architecture)

Software-only memory protection solutions are generally too slow (400%-1000% overhead). Hardware-assistance is needed.

Market maturation on-going
(in all categores: IoT, Consumer, Cloud)



Vulnerabilities

70% of the attack surface in Linux kernel, browsers, etc. relies on some form of memory vulnerability being exploited

This is different from 15 years ago, when the attacks typically consisted of overwriting code. MMU mechanisms like memory W^X has added strict separation between code and data

Instead, techniques like return-oriented programming (ROP) have become dominant.

"Eternal war in memory", Dawn Song & al, 2013

https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf

Memory protection features are used for
1) Isolation of workloads (this is nothing new)
2) Guaranteeing execution integrity - that the program behaves as the programmer expects

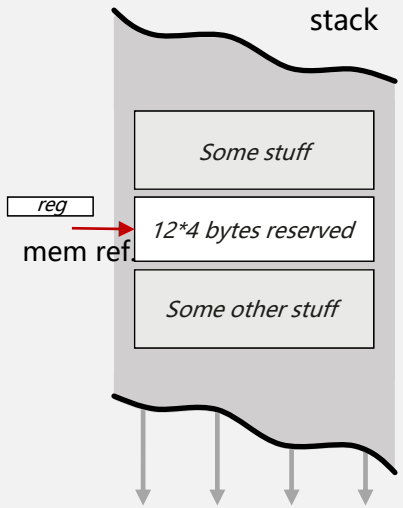
Main topic of this lecture

Run-Time Memory Protection -- the How and Why

C/C++

```
int i, j, a[16];
..
a[i++] = j;
```

compile



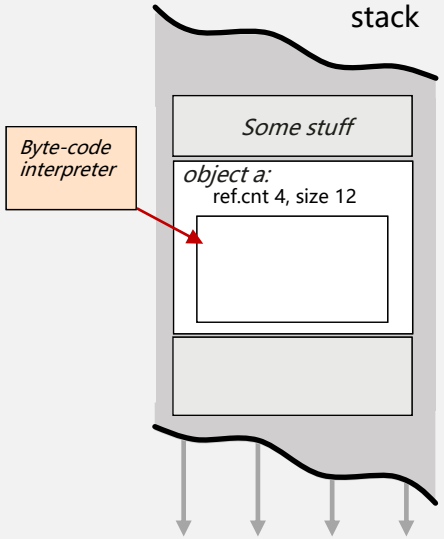
***a** does not exist any more all type information is lost only address remains.*

Memory unsafe setting

Java

```
int i, j, a[16];
..
a[i++] = j;
```

Compile to Java bytecode

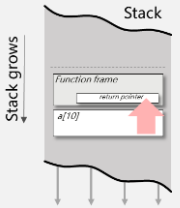


***a** is still known, boundaries can be enforced and reference counts tracked*

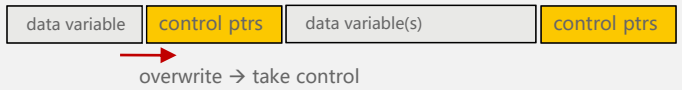
Memory safe setting

.. so what is happening in an attack ?

Microsoft 2021: 70% of all exploitable software vulnerabilities are like this → memory vulnerabilities



Linear memory



Problems:

- ✓ Control structures and data are mixed in linear memory
- ✓ Bugs in code exist
- ✓ Bugs can be exploited

Attack:

- ✓ Exploit a bug
- ✓ Overwrite control data with e.g. pointer value of choice
- ✓ Take code control or leak information

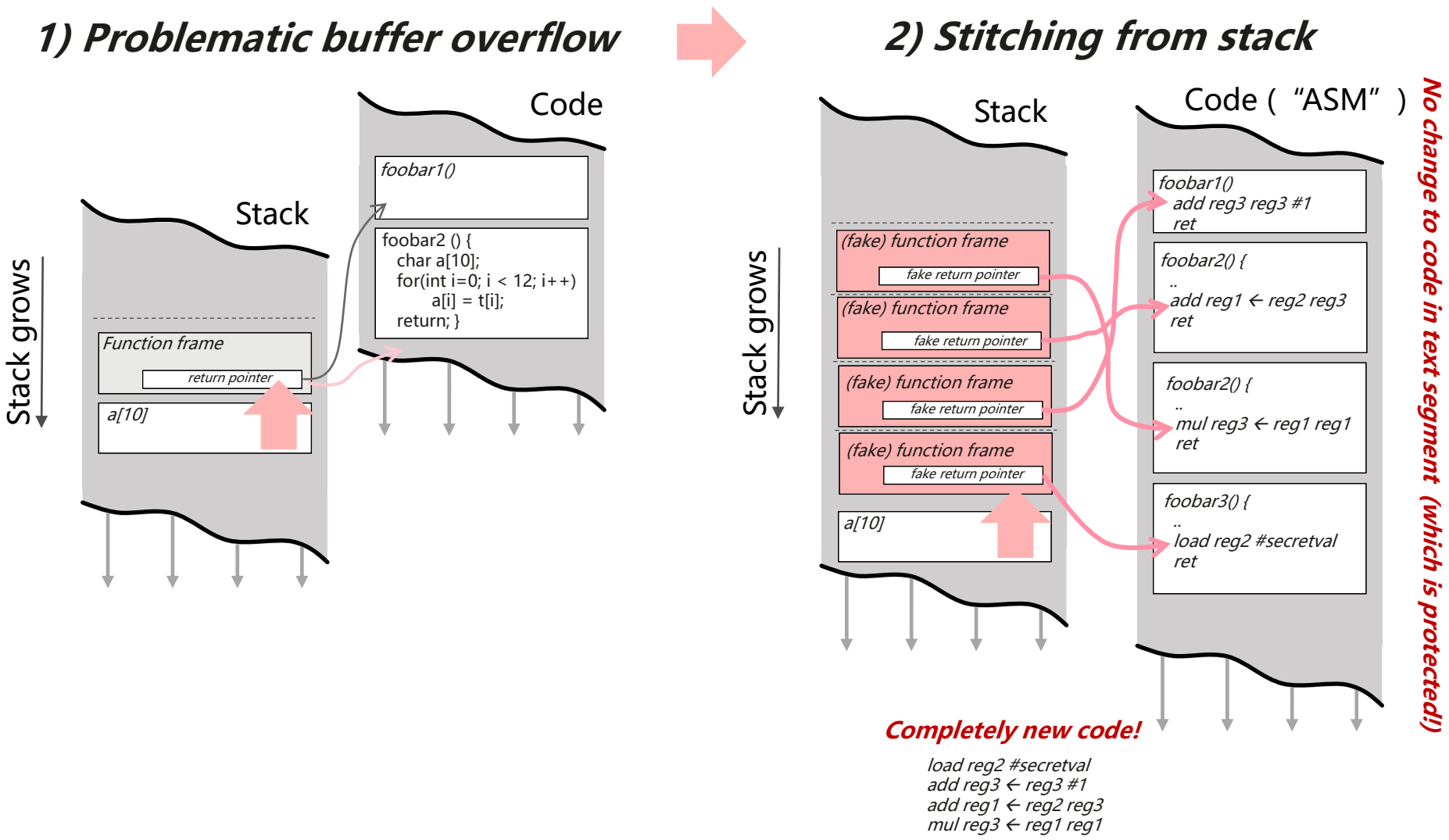
Run-time protection is what we can do during run-time to mitigate intentional or unintentional bugs turning into vulnerabilities and attacks:

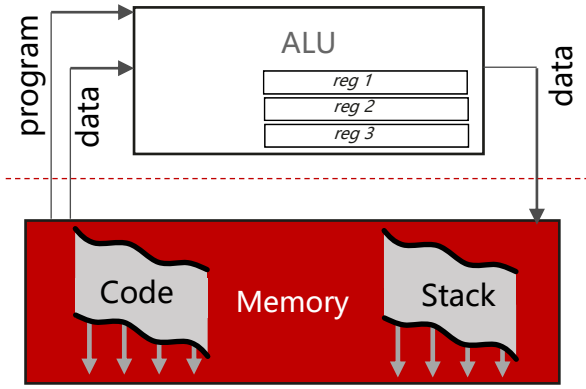
- Formal proofs and model-checking to validate that code has no bugs. In essence, use better programming languages
- VM-based execution(script languages) to sandbox execution, and limit attack surface
- Compilers and security tools can a) statically identify problems, or add protections in software
- Leverage new architectural features in hardware (with compiler assistance) to add protection to memory-unsafe execution

Presentation idea adapted from a talk by prof. Elias Athanasopoulos, University of Cyprus

The problem(s) in fine-grained memory protection …

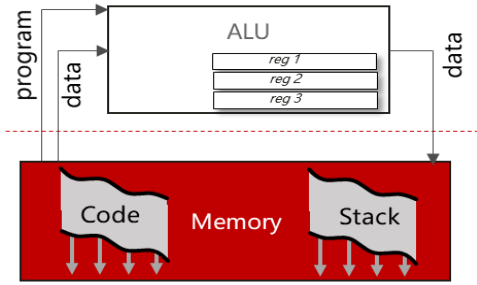
… comes in countless variants, but here an abstraction of "Return-Oriented Programming" (ROP)





- Attacks often ***Turing Complete***
- Can be found on all architectures
- Automated stitching tools and gadget finding
- Violates call-flow (protections often called Call Flow Integrity (CFI))
- But there are others, e.g. Data-Oriented-Programming (DOP)

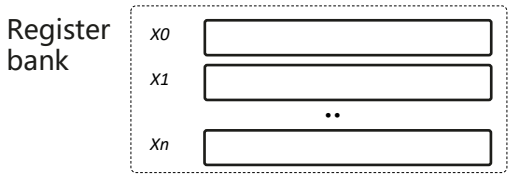
Running programs in a computer – the relationship between language, compiler and binary code



- The instruction code semantics in most legacy CISC / RISC computers today are surprisingly similar to each other
- One abstraction that is often used to describe the machine instruction set is **p-code** (originally a machine-independent byte-code for Pascal [1970s]) E.g. 'Algorithms + Data Structures = Program´ [1976]

Registers:

- Data buffer close to the ALU
- Banked registers change content by context , e.g. based on privilege level
- General purpose registers are registers usable for arithmetic (source, target, e.g. x = ADD y, z)
- Memory mapped registers reside in a peripheral, and are accessed via a memory pointer
- The stack pointer is a register that tells where (in memory) the top of the stack is
- The program counter is a register that tells where (in memory) we are currently executing code



(Machine) instructions:

- LOAD / STORE from/to memory
- Perform arithmetic (mainly between registers) ADD / EQ / MUL / XOR , ..
- LOAD immediate value
- Conditional Jumps → move PC (JNE, ..)
- SUB / RET → Some way to jump to a subroutine and to RETURN to the place you called from.

The main categories of machine instructions are very few, and essentially type-less (no notion of char vs int vs ..)

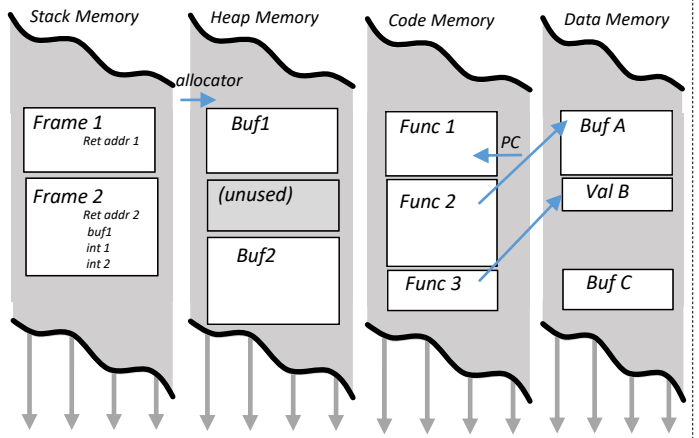
All other functionalities are built using these primitives. Extensions may include floating point, vectors, crypto, AI

CISC: Complex instruction Set (X86)
RISC: Reduced Instruction Set (ARM)

Memory organization:

- .text = Executable
  .heap = Data reserved dynamically
  .data = Static data declared in code
  .stack = Compiler –assisted organization for local variables and call-flow control data

The memory organization is a strong convention, but not a necessity. However, for security, we will see that providing proper access control for right kinds of memory is of great benefit (e.g. W^X)

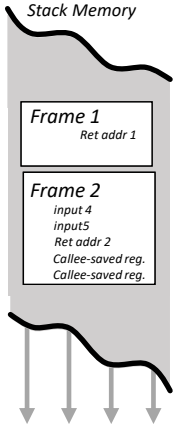


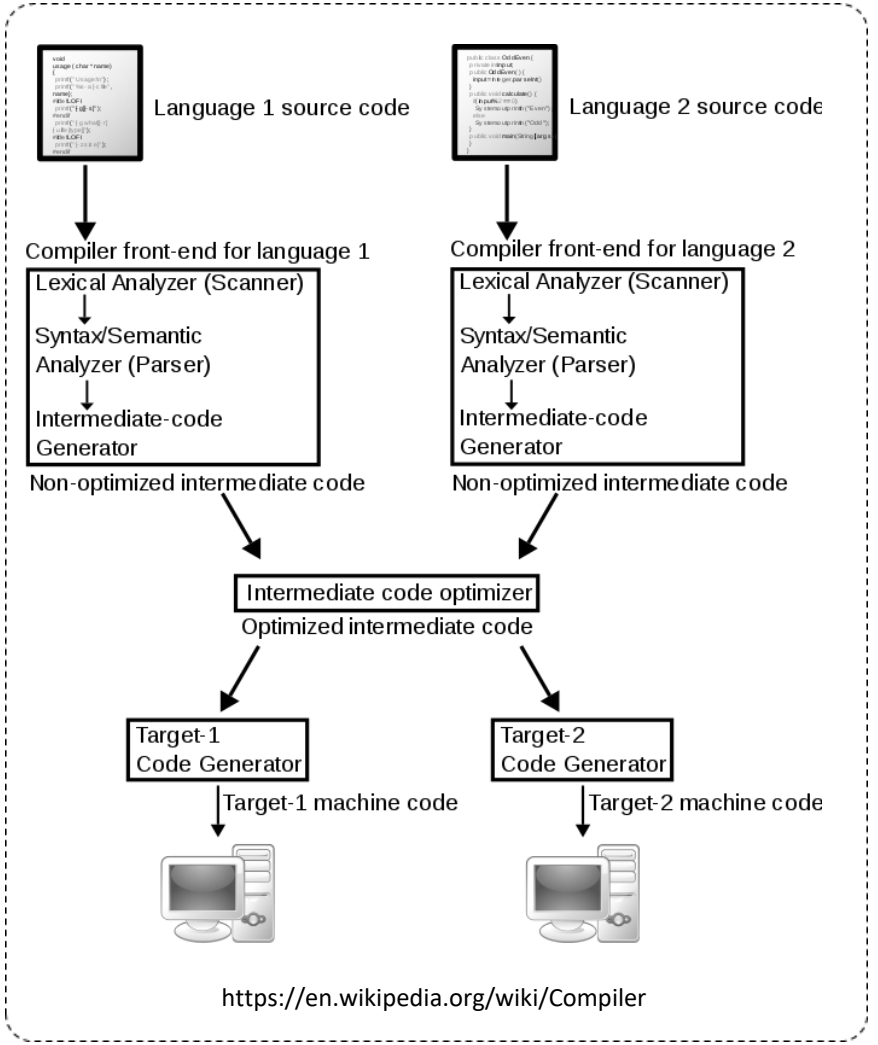
The Role of the Compiler for Security

- The most prevalent system compilers today (for C/C++) are GCC and the LLVM compiler suite. It seems that LLVM is gaining traction, and most new research and development happens on the LLVM initially. But GCC has soon a 40-year history, and the biggest supported set of target machines
- The output of compiler has to adhere to the ABI (Application Binary interface), which defines things like how to call a function, how to load a dynamic library etc.
- For security, the optimization passes invoked after the lexical and syntactic passes (AST generation), are often important. This is the place where additional, compiler-introduced protection mechanisms and / or static analysis for security can take place
- The back-end, or code-generation stage of the compiler of course needs to be modified, if the target hardware provides architecture, mechanisms or new instruction for (memory) security

ABI

- When calling a function, which parameters (call + return) are passed in registers, and which on the stack frame?
- When registers are needed in the called function, whose task is it to back-up contents? Caller-saved registers vs. Callee-saved registers.
- Memory alignment for different types. Composite parameter call by value or by reference, (This is a language feature, but has hardware implication e.g. for floating point.
- Variadic functions, e.g. functions with variable number of arguments (e.g. printf)
- Exception unwinding (setjmp / longjmp).





https://en.wikipedia.org/wiki/Compiler

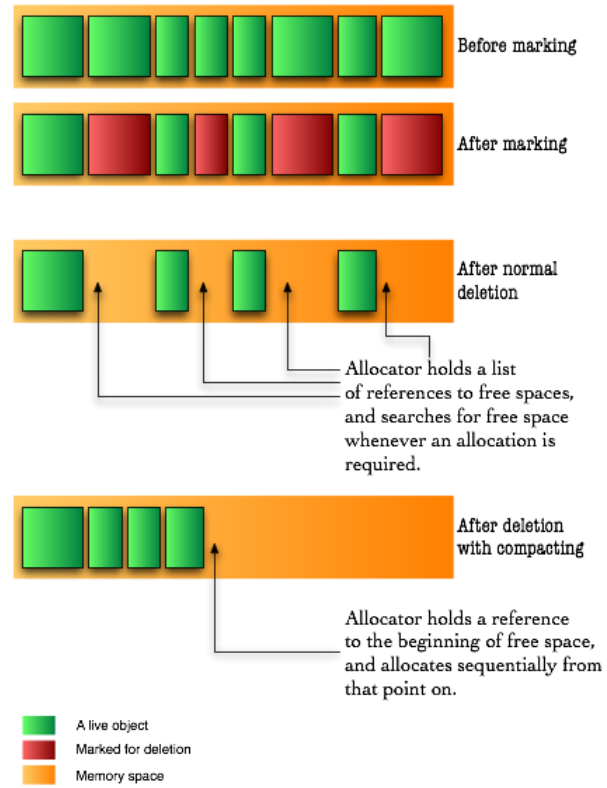
Compiler Mitigations (Software)

From last week´s lecture recall

- Address Space Layout Randomization (ASLR, KASLR)
- Canaries (we will return to these)
- W ^ X : memory protection for stack, data, heap vs code
- UXN / PXN : Unpriviliged / Privileged Execute Never (kernel interface protection)

Garbage Collection

- **Basic idea**: Every reference to an allocation must be tracked, so that the allocation can be freed when no references to it are left. This can be implemented in a language runtime, in the compiler, in hardware or any combination of these
- **Reference counting**: Each allocation has a counter attached, containing the number of references to it.
  - The reference count is incremented when a new reference is created, and decremented when a reference is destroyed.
  - When the reference count reaches zero, the system knows that the allocation is accessible, and can be freed.
- **Tracing garbage collection**: The garbage collector starts with a set of "root" objects (e.g. the main class), and follows their references to find all reachable objects.
  - Old approach: "mark+sweep" occasionally pauses the program, traverses the whole tree, and then frees any allocations not seen during traversal. Since the program is paused, it can also move objects around to reduce memory fragmentation ("mark+compact")
  - Newer GCs can run concurrently with the program, reducing pauses.
- Some sort of garbage collection is used in many modern computer languages (Java is the oldest widespread one, but also Python, Go, Javascript, Rust [optionally]). Even C-solutions (for heap) exist, e.g. Linux kref, Symbian (Epoc).
- **Issues:**
  - Real-timeness is hard to achieve together with GC
  - For system programming, references and memory access is not always program-defined -- I/O addresses, DMA etc. This easily conflict with program-assisted memory management



https://stackoverflow.com/questions/4813005/garbage-collection-java

CHERI garbage collection: Cornucopia

https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9152640
https://dl.acm.org/doi/pdf/10.1145/3352460.3358288

Symbian Memory Management

https://www.slideshare.net/andreasjakl/symbian-os-memory-management

Jones, "The Garbage Collection Handbook", 2016.

Fortify-Source

- Fortify-source is a buffer-overflow check, that is more or less implemented as a pre-processor or front-end addition to the compiler, using the –D_FORTIFY_SOURCE=val flag. The mechanism is not fool-proof, but aims to protect against buffer overflows.
- **Basic idea**: For many allocated buffers (e.g., in stack, the compiler knows the size of the buffer during compilation - say for a declared buffer ‘unsigned char buf[7].
  - There are a number of buffer / string handling functions (variants) that can check against overruns, including memcpy, memmove, memset, snprintf, sprint, strcat, strcpy etc. These are not always used in code.
  - Based on simple static analysis, the fortify_source changes the compiled program to use these boundary checking variants of the functions, causing the program to crash in case of buffer overruns
- The solution can be applied on allocation level, or even intra-allocation boundaries can be respected (but some codes intentionally violate those)
- **Issues:**
  - The strength of the mechanism is hard to quantify, primarily it fixes programming errors that could / should(?) be done as a one-shot activity
  - The analysis does not necessarily carry far, e.g. for pointers derived from an allocation

Example reproduced from RedHat blog

Declaration

```
char buf[5]
```

Statically declared safe, no change

```
memcpy (buf, foo, 5);
strcpy (buf, "abcd");
```

Length of copy unclear, use checking function variant

```
memcpy (buf, foo, n);
strcpy (buf, bar);
```

Buffer length unknown, no protection

```
memcpy (p, q, n);
strcpy (p, q);
```

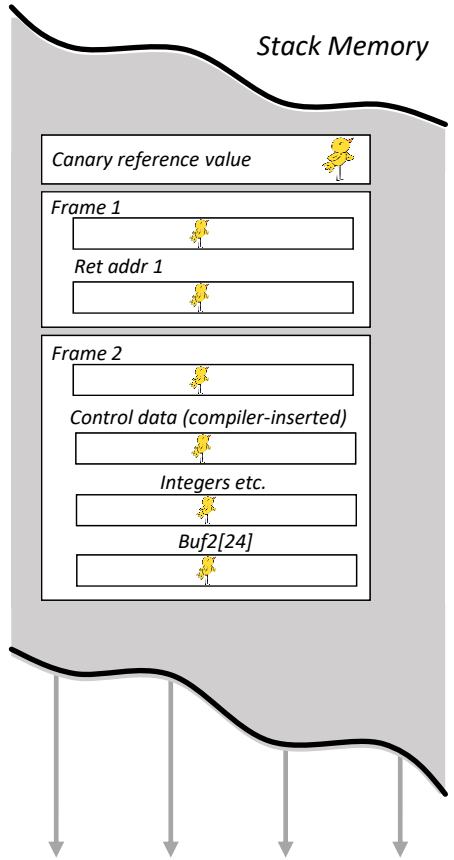
Intra-allocation case (buffers can be protected based on S or buf)

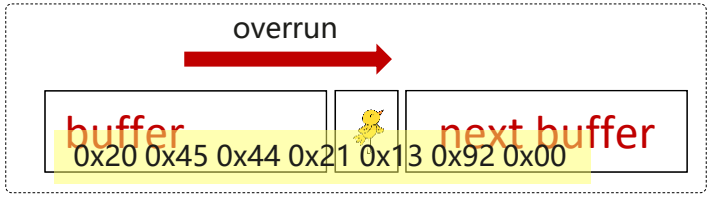
```
struct S {
   struct T {
      char buf[5];
       int x;
       } t;
    char buf[20];
} var;
```

https://www.redhat.com/en/blog/security-technologies-fortifysource?extIdCarryOver=true&sc_cid=701f2000001OH6pAAG
https://developers.redhat.com/blog/2020/02/11/toward-_fortify_source-parity-between-clang-and-gcc

Stack Canaries

- In LLVM, this feature can be enabled by **–fstack-protector** and friends. Here shown for stack, but principle can of course also be applied for .data (or with dynamic support for heap)
- **Basic idea**: On every run, decide on a canary value, e.g., a random value. It can be program-wide, thread-specific, even specific to function frames. But the mechanism must be efficient also
  - When a function frame is constructed, compiler writes the canary value between segments of the frame that may have overflow issues (e.g. character buffers)
  - When the function terminates, the compiler adds checking code, i.e. before returning this code checks that the canaries are unmodified
  - The understanding of this security feature is that the canary value is unknown to the attacker, and therefore a buffer overflow will cause canary values to be different from the expected one, which will be exposed at function return.
- **Issues:**
  - Reference canary values are most often in memory, and can be exposed / overwritten
  - Protect against linear overruns, but not against overruns that do not touch consecutive memory addresses
  - Check is not immediate – attacker may correct before check
  - Stack-based alloca:s, can reserve and free within the boundary of the function. Also, functions can be very long-lasting (whereby check never happens)





https://developers.redhat.com/articles/2022/06/02/use-compiler-flags-stack-protection-gcc-and-clang#stack_canary

Safe Stack (Shadow Stack)

- In LLVM, this feature can be enabled by **-fsanitize=safe-stack** and this creates a software shadow stack, instrumented by compiler. Hardware assistance for this feature is available as part of Intel CET and the recently announced ARM shadow stack

- **Basic idea**: one shortcoming of the stack as use by most languages, is that control data (function returns and other stack-frame related data (frame pointer previous stack pointer) are stored on the stack interleaved with function local data that can be vulnerable to e.g. buffer overflows. But, what if

  - All the control data is stored in a separate stack, not interleaved with buffer accessible by the program

  - The new stack is either well hidden (so that its address cannot be resolved, and intentional overwrites done) or isolated by hardware so that its operations as a stack are well governed by hardware architecture

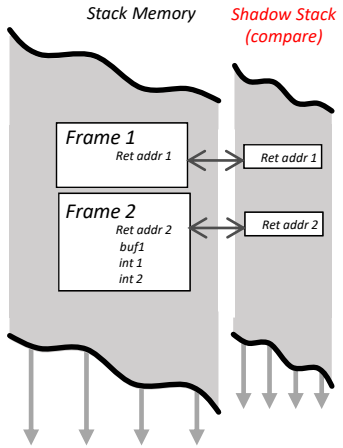
  In this case much of the call-flow (especially function returns) can be protected against attacks

- **Issues:**

  - Without hardware support, it is almost impossible to hide the location of the shadow stack, opening the possibility for new kinds of vulnerabilities

  - To note is that a shadow stack by default does not address indirect forward calls (vtables in C++ or similar). For that, other solutions are needed

Shadow stack principle



LLVM Safe Stack https://clang.llvm.org/docs/SafeStack.html

- SafeStack protects return addresses, spilled registers and local variables that are always accessed in a safe way by separating them in a dedicated safe stack region.

- The safe stack is allocated at a random address and the instrumentation ensures that no pointers to the safe stack are ever stored outside of the safe stack itself (exception handling is still a problem)

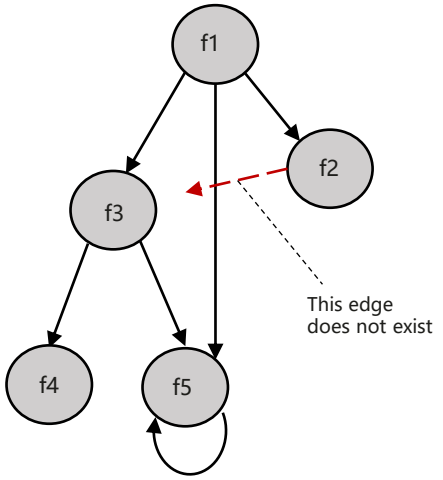
Average overhead of shadow stack can be as little as 0.1%

https://dslab.epfl.ch/pubs/cpi.pdf

CFI – Call-flow Integrity

- Call-flow integrity is a general security intent, where we want to assure that the call-flow between software units in our code (read functions, object methods, library calls) happens as is intended by the programmer. This includes function returns as well as direct and indirect (via memory) forward jumps. Typically the indirect calls are the most difficult to protect (direct calls are safe, if code adheres to W^X)
- Shadow stacks is the default, efficient protection for returns. However, there is no easy-to use protection for indirect calls, since a) many languages allows forward calls to be dynamic (decided on during run-time) and b) there is a large mismatch between call-sites for a given function (say memcpy can be called at 1000´s of locations, many other functions only by 1-3 locations). Vice-versa, some functions call a large number of different functions (even more or less at the same location), some are more static in that respect.
- The reference for which function calls other ones is called call-flow-graph, and this contains the call-flow transitions that can be statically inferred by the compiler 8excluding intended run-time behavior).

Call-flow Graph



LLVM Software CFI

- Included by **–fsanitize=cfi.** This mechanism include a large variety of type-based independent protections for various types of forward-edge indirect calls, below only an idea is given for some of these:
- Indirect function call type checking: The dynamic type of the function must match the static type of the call. I.e. the compiler adds code enforces that enforces prototype checking at run-time
- Bad-pointer-cast protection, e.g. from a base class to a derived class in C++, or from void * to a function with type. Some of this is not necessarily according to the C/C++ standard, so software changes might be needed.
- Forward-edge CFI for virtual calls, i.e. enforcement that he dynamic type of the called object must be a derived class of the static type of the object used to make the call

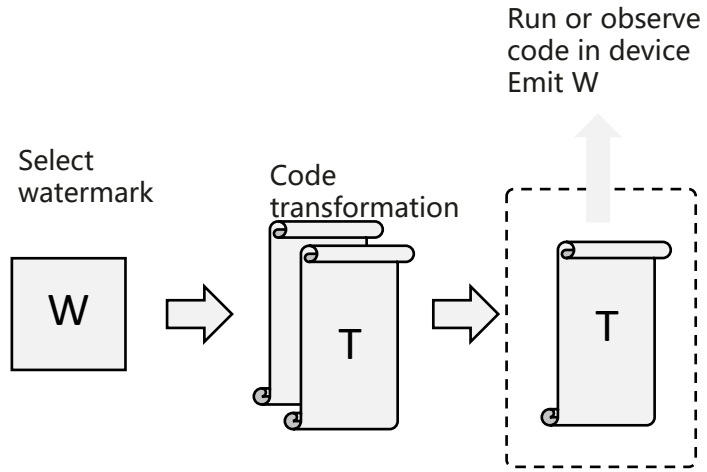
https://clang.llvm.org/docs/ControlFlowIntegrity.html
https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html
https://arxiv.org/pdf/1910.01485.pdf
https://patrickfunke.de/control-flow-integrity-cfi-for-gcc-llvm/

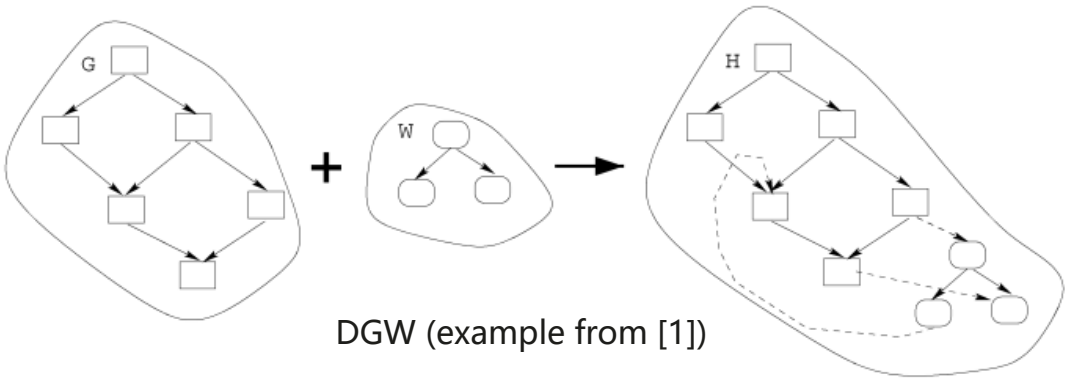
Watermarking (Research)

- The compiler can also be augmented to do code watermarking. This kind of approach allows compiled software to be identified after deployment e.g. to identify piracy. Watermarks can take many forms – a license string (easily removed), an easter egg function responding to a secret input (exposing the origin) or simply a controlled re-ordering of the code (modules, functions, local variables etc.) based on a identifier input (static watermarking). Even CFGs can be adapted to watermarking

- Watermarks are subject to attacks, if an attacker gets hold of two or more binaries with watermarking. Attacks can be additive (an extra watermark is added by attacker to put doubt on an original, unfound watermark), Subtractive attacks where watermarks are removed based on difference in two images and distortion attacks, where the binary is further re-ordered to distort the watermark efficiency.

- **Basic idea**: The watermarking can be done in several distinct ways:
  - In ´source' , e.g. by re-ordering local variables so that their location in the stack frame depends on the watermarking input
  - By data transformation (in practice close to obfuscation), where the selected data transformation depends on the input. The benefit of such methods is that removal based on code differences might be difficult. See example
  - Dynamic graph watermarking: Select a value W (watermark); Embed W by selecting the W:th graph in a particular enumeration of a particular class of graphs. Make the run-time of the program (e.g. call-flow) produce this graph with a particular input





DGW (example from [1])

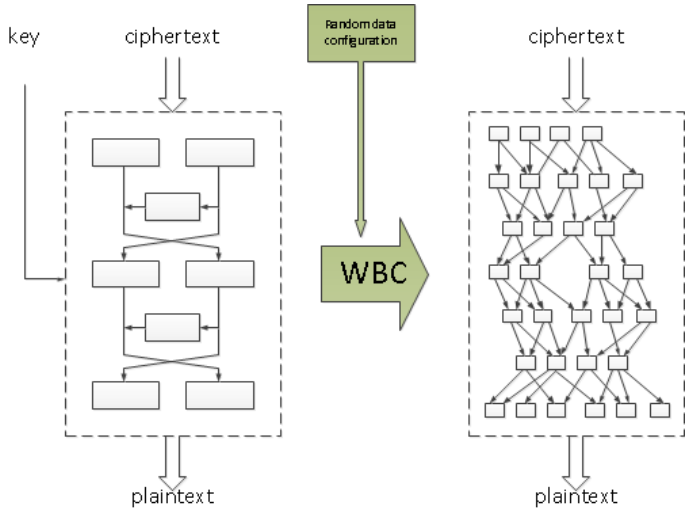
Data transformation with equivalence (example from [1])

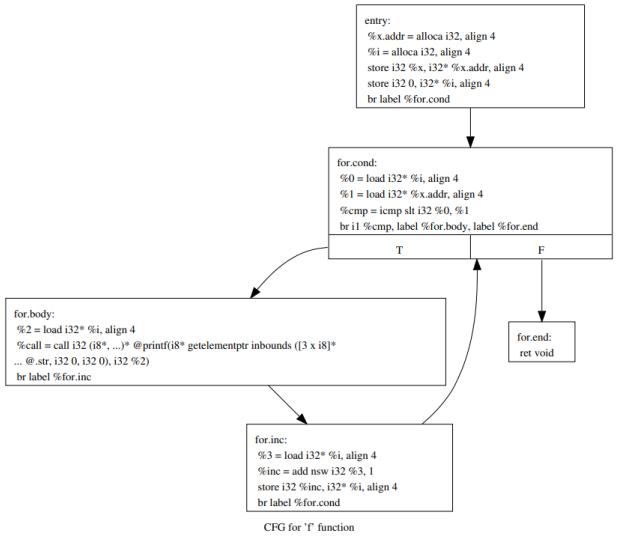
```
(1) bool A,B,C;          (1') short a1,a2,b1,b2,c1,c2;
(2) B = False;           (2') b1=0; b2=0;
(3) C = False;     ⇒     (3') c1=1; c2=1;
(4) C = A & B;           (4') x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(5) C = A & B;           (5') c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(6) if (A) ···;          (6') x=2*a1+a2; if ((x==1) || (x==2)) ···;
(7) if (B) ···;          (7') if (b1 ^ b2) ···;
```

[1] https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1027797

Obfuscation (Research)

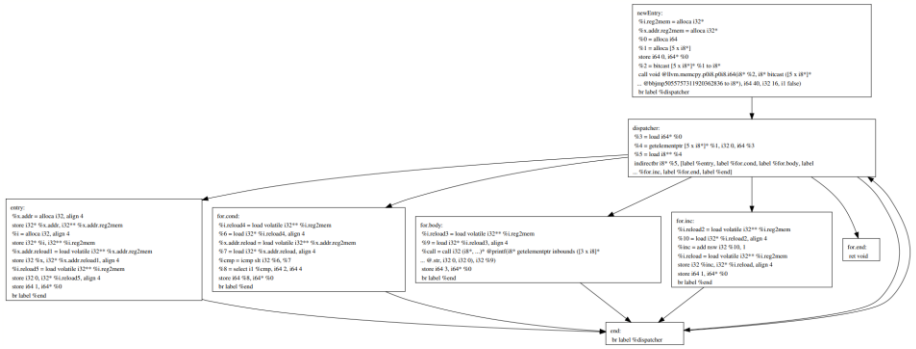
- Obfuscation is primarily done to prevent reverse-engineering, but can also be applied to implement software-only isolation, e.g. things like white-box cryptography. By applying control transformations and data transformations, by removing program information from functions and objects (e.g. useful for interpreted code like Java), and even self-transforming code (mutation), to make analyzing the program (from binary) difficult.
- Computer viruses also use obfuscation techniques to hide their presence in carrier programs. DRM is another use case, or to hide control structures that might want to be removed by the users (say in-app purchases) in existing code.
- Obfuscation can be done as a pre-processing step, but especially call-flow transformations are most easily done as part of a compiler.





CFG for 'f' function

From [1]: Example function in LLVM IR (orig)



Same function with CF obfuscation + flattening

```
int a,b,c;
clrscr();
printf("Enter two no.:");
scanf("%d%d",&a,&b);
c=a+b;
printf("The Addition is:%d",c);
```

--Original.c--

```
int main()
{
unsigned long a,b;

unsigned long m=12;
unsigned long s=0x20;
unsigned long ae,be,x;
printf("Enter two no.:");
scanf("%d%d",&a,&b);
ae=a^m;
be=b^m;
printf("%lu %lu\n", ae,be);
x=(ae | be<<s);
printf("%lu\n", (x&0xffff)
+ (x>>0x20));
return 0; }
```

--Obfuscated.c--

[1] https://crypto.junod.info/spro15.pdf
[2] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9167661
[3] https://www2.seas.gwu.edu/~simha/research/SoftwareSecurityBookChapter2012.pdf
[4] http://www.whiteboxcrypto.com/
[5] https://pages.cs.wisc.edu/~arinib/writeup.pdf

Address Sanitization (Software Testing)

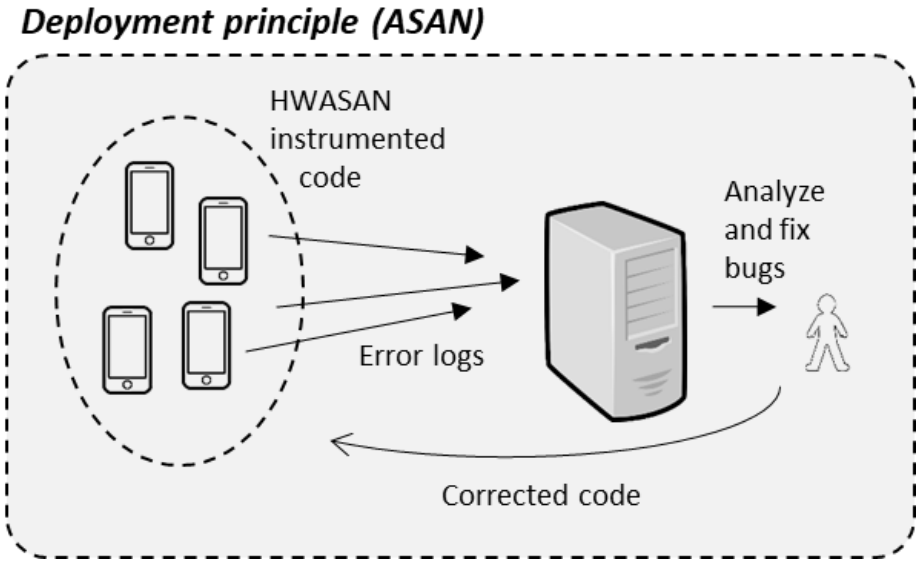
- An address sanitization framework is a way to test a program against memory errors before deployment. In its core, this amounts to a debug compilation with as much type information about objects and run-time memory reservations stored as possible, + a run-time library to follow the dynamic use of memory during the run of the program.

- The LLVM flag to enable this is **–fsanitize-address** . For LLVM, the additions to the instrumented binary are significant, and the expectations on memory usage (and speed) overhead is around 100%

- Address sanitization should be able to find
  - **Out of bounds accesses** to heap, stack and globals
  - **Use after free**
  - **Use-after return** (stack object used after stack was unwound)
  - **Use after scope** (variable used outside scope of the variable)
  - **Invalid / double free** (freeing already freed memory)

  - Some variants of **memory leaks**, i.e. reserved memory to which the program loses reference

- There is still the issue of coverage: Can the software be tested fully (based on all possible inputs) to make sure that all memory problems are uncovered during testing? Therefore, a larger testing community is good to establish.

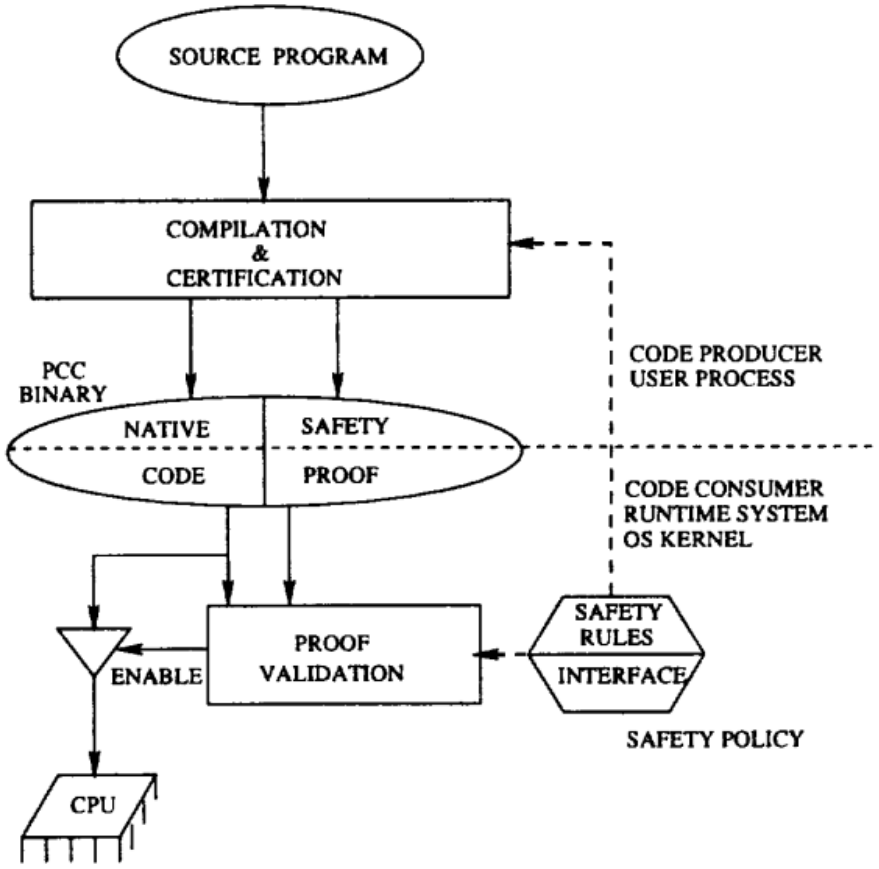


Proof-Carrying Code (Research)

- A compiler research direction that has existed since the 90´s (Necula: Proof Carrying Code). The main idea is that the compiler emits a formal proof about the safety of the program, which then can be validated in the device by a proof-validating daemon, and following the device´s policy / ruleset, the device can verify the proof and how it matches the policy before the program execution is initiated.

- In is early instantiations, the daemon was actually a theorem prover, i.e. the received code was through a formal proof carried our in the device before the code was run. This has the advantage of being integrity-preserving, **in that any modification in the program either satisfies the proof (whereby it is OK to run) or does not (whereby the program is not launched).**

- However, e.g. in consumer devices, having a theorem prover handy seems like an overkill. For that, the architecture can be modified, and a "certifying compiler" added, i.e. the compiler would conduct all necessary proof generations (possible with the help of the developer) and then cryptographically certify the properties that were proved – minimizing the device-end work to safety feature matching.

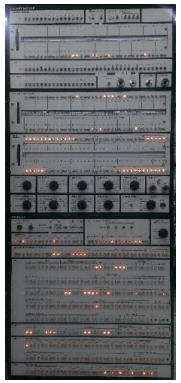


[1] https://dl.acm.org/doi/pdf/10.1145/263699.263712
https://link.springer.com/content/pdf/10.1007/1-4020-8141-3_27.pdf

Hardware Mitigations

1. The Hardware Solutions
2. How to Use Them (Compiler)

Multics 6180



http://ana-3.lcs.mit.edu/~jnc/tech/multics/MulticsPanels.html

Secure memory domains

1979, IBM

IBM System/38

http://bitsavers.org/pdf/ibm/system38/IBM_System_38_Technical_Developments_Dec1978.pdf

Hardware capabilites

40 years of nothing

Sparc M7

https://source-tech.net/shop/new-servers/sun-oracle-servers-new-servers/sun-oracle-sparc-t7-2-server/

2016, Oracle

Memory coloring

!

Hardware Protection -- History

Solutions dedicated to solving contemporary memory protection issues (intel)

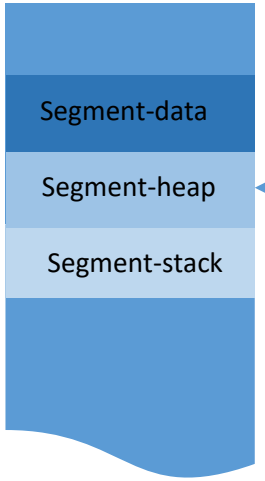
Hardware segments

- ★ First in MULTICS
- ★ Still visible in i286 / Win 3.1

Good memory separation, clumsy to operate, compile

Course granule

Ptr: SEG-register + segment offset

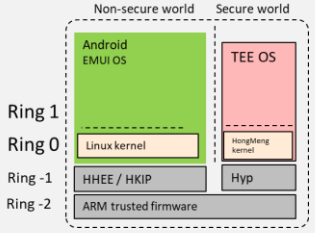


Hardware rings / hardware domains (TZ)

- ★ First in MULTICS
- ★ In significant security use especially on ARM

Protects domains from each other, hyp / kernel / user-space separation

Efficient specifically on macro-level



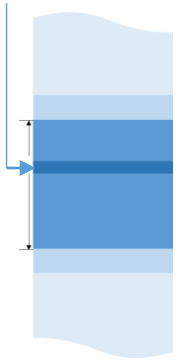
Hardware capabilities

- ★ First in Cambridge CAP computer
- ★ Not visible in contemporary devices

Fine-grained access control + boundary checks

Efficient, small granule

Ptr: base + limit



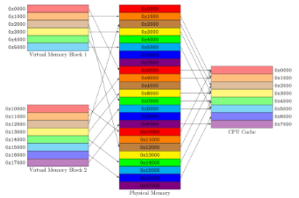
Hardware tags / coloring

- ★ First in Burroughs (1970s, later in Sparc -- 2010s)
- ★ 'Displaced' by virtual memory)

Temporal safety for bounds checking, memory reference type support

Efficient, restricted by tag resolution

*)



Privilege domains + VM

Main (only) HW Security solution 1985-2015

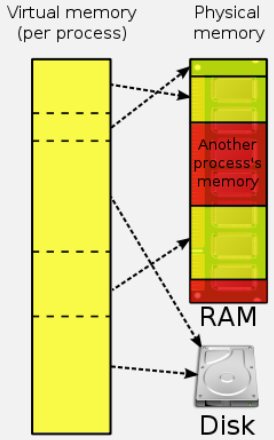
Granularity: 1 page (+ banked regs)

Hardware address translation

- ★ First in VAX/VMS (~1980)
- ★ Omnipresent in contemporary architectures

Page-based access control, cache support

Effective for e.g. process isolation



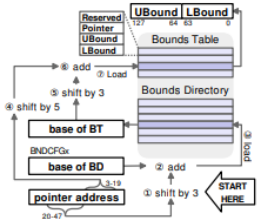
Code-controlled bounds checking

- ★ Intel MPX (2017-)
- ★ Very low overhead

User code-operated, stacked bounds enforcement

Bounds checking with non-fat pointers

**)



Memory Protection Keys for UserSpace

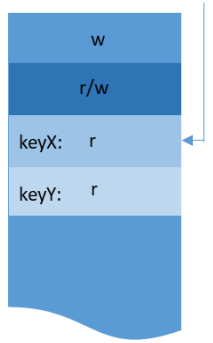
- ★ Intel MKU (Skylake ->)
- ★ First iteration IBM/360, in 60s but then for task isolation

User code-operated, page-based access control domains

Effective for e.g. rare-write protection

***)

```
pkey_set(keyX, 0); // enable write
*ptr = foo; // assign something
pkey_set(keyX, PKEY_DISABLE_WRITE);
```



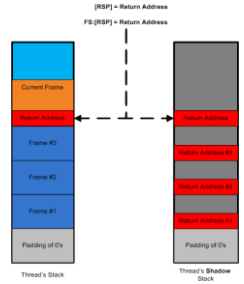
Hardware shadow stack

- ★ Intel CET (2019-)
- ★ To-be deployed

Hardware shadow stack, providing full return edge CFI,no overhead

Dedicated primitive for important CFI prot.

****)



1960 — 1970 — 1980 — 2015

*) http://www.feustel.us/Feustel%20&%20Associates/Advantages.pdf

**) https://intel-mpx.github.io/code/submission.pdf

***)https://www.kernel.org/doc/Documentation/x86/protection-keys.txt

****) https://eyalitkin.wordpress.com/2017/08/18/bypassing-return-flow-guard-rfg/

Break

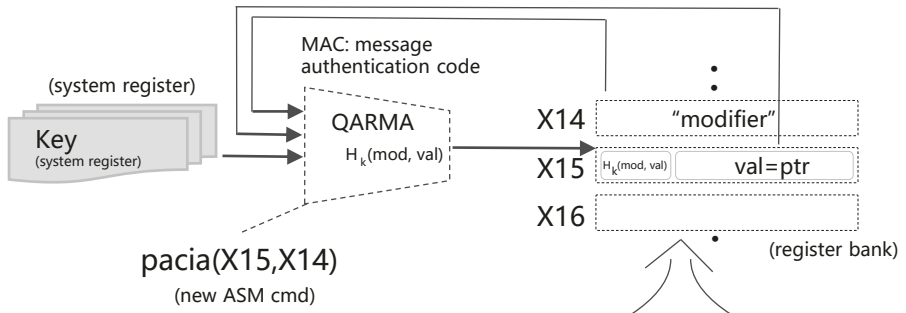
(if time permits)

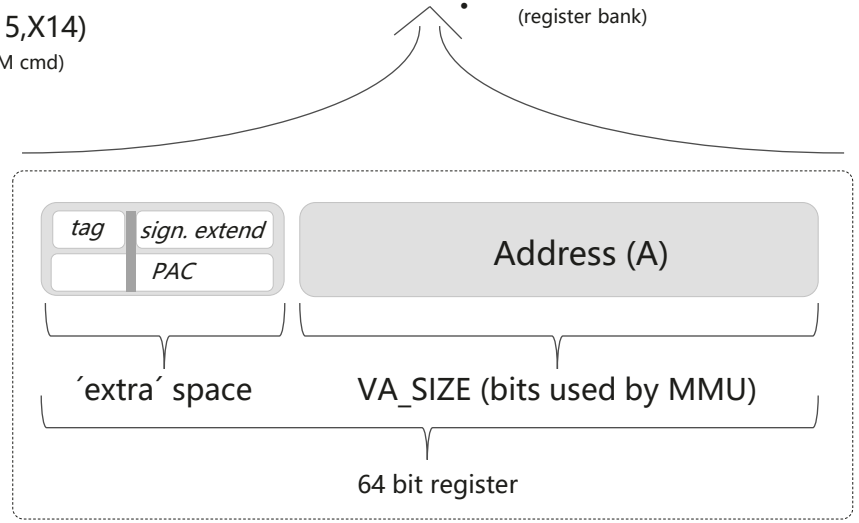
ARM Memory Protection Features 2019→ (ARMA-v8.3 - ARMAv9.x)

- These hardware extensions are all general-purpose solutions to achieve memory protection: To integrity-protect stores and loads (PA), to label memory and references to it to e.g. protect overflows, and to harden call flow targets to e.g. circumvent ROP attacks

① *Pointer integrity*
- digitally sign pointers put in memory

ARM Pointer Authentication (ARMAv8.3 → 9.x)

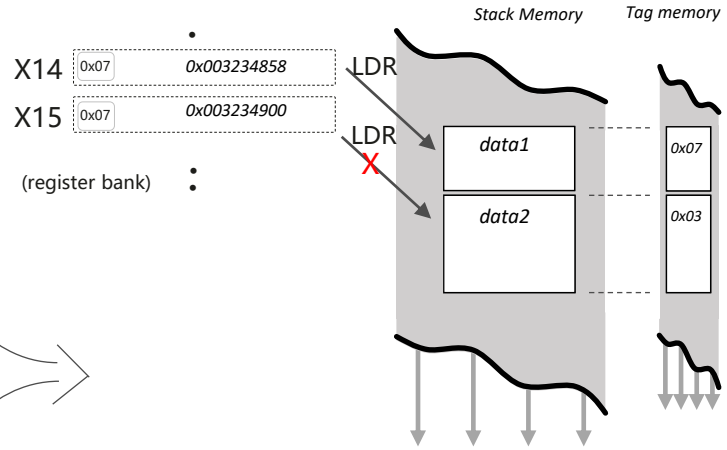




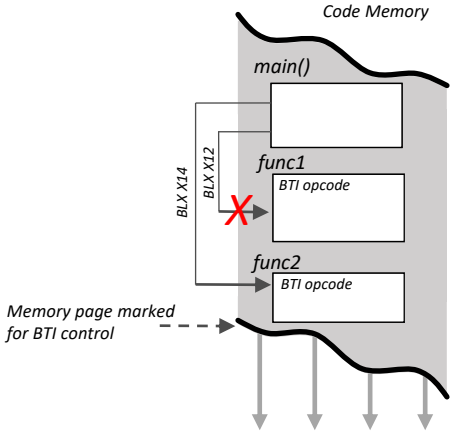
② *Color memory and pointers*
- Access only allowed if memory color and pointer color match

Set tag value in memory
STG (X15,X16,#0)
(new ASM cmd)

ARM Memory Tagging (ARMAv8.5 → 9.x)



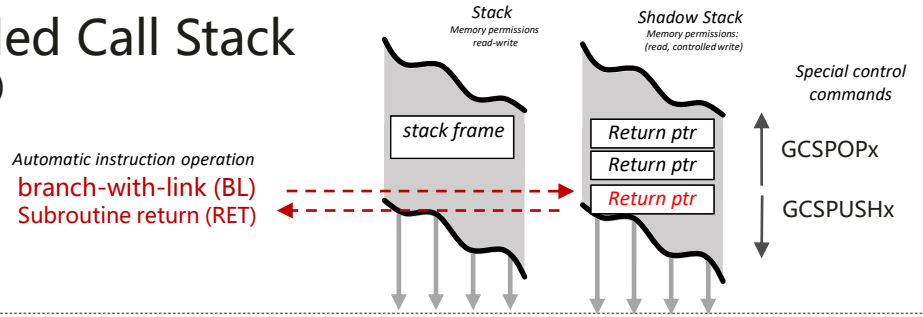
ARM Branch Target Integrity (ARMAv8.5 → 9.x)



③ *Constrain jump targets*
- Limit jumps to beginnings of functions (not possible to jump into the middle of a function)

④ *Enforce control flow over function calls*
- ´remove´ return address memory storage from stack

ARM Guarded Call Stack (2022 extensions)



Intel Memory Protection Features 2015→

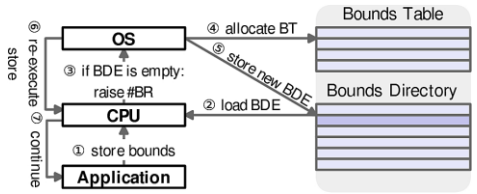
Intel MPX (Sky Lake →)
Intel memory protection extension

- 4 new registers for bounds

(register bank)

bnd0	Ubound0 / Lbound0
bnd1	Ubound1 / Lbound1
..	

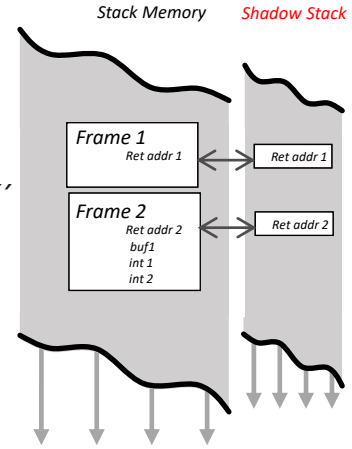
- Pointers in registers are associated with bounds checking using a set of MPX instructions. Allocation of bounds intended to be the done by OS, checking added by compiler
- Bounds violations are a new form of trap / interrupt that need to be handled
- Architecture for maintaining bounds in memory done in HW (can be expensive for performance)



From: https://intel-mpx.github.io/design/

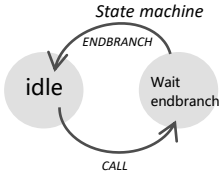
Intel CET (Tiger Lake →)
Intel Control Flow Enforcement Technology

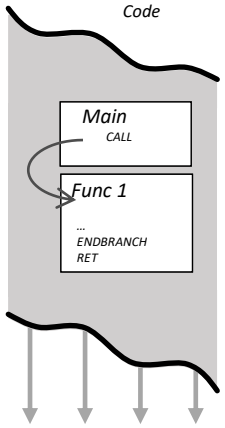
- The return address in the function frame is double-checked against a ´shadow´´ value orchestrated by hardware
- New instructions for unwinding shadow stack and context switches



Indirect Branch Tracking

- A HW state machine that enforces alternating indirect JMPs and new ENDBRANCH instr.

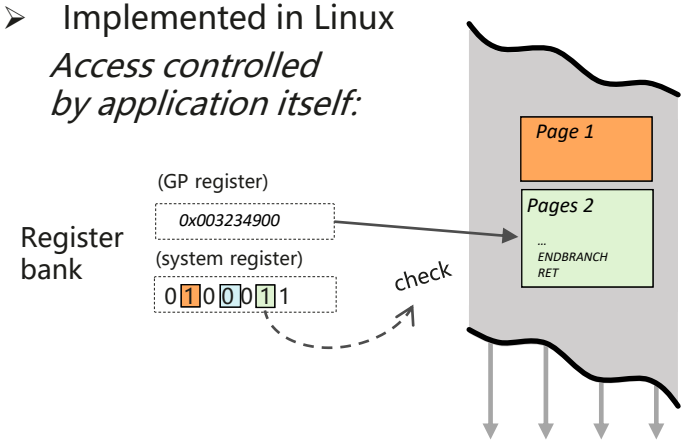




Intel MPK (Sky Lake →)
Intel Control Flow Enforcement Technology

- A memory-page based control, where colors in a 32-bit user-space register shall match a memory-page color in order for memory access to work
- Although no different from VM page protection, the access can be controlled from within application with little TLB impact – allows for efficient and fast access switching for applications that need this feature
- Implemented in Linux

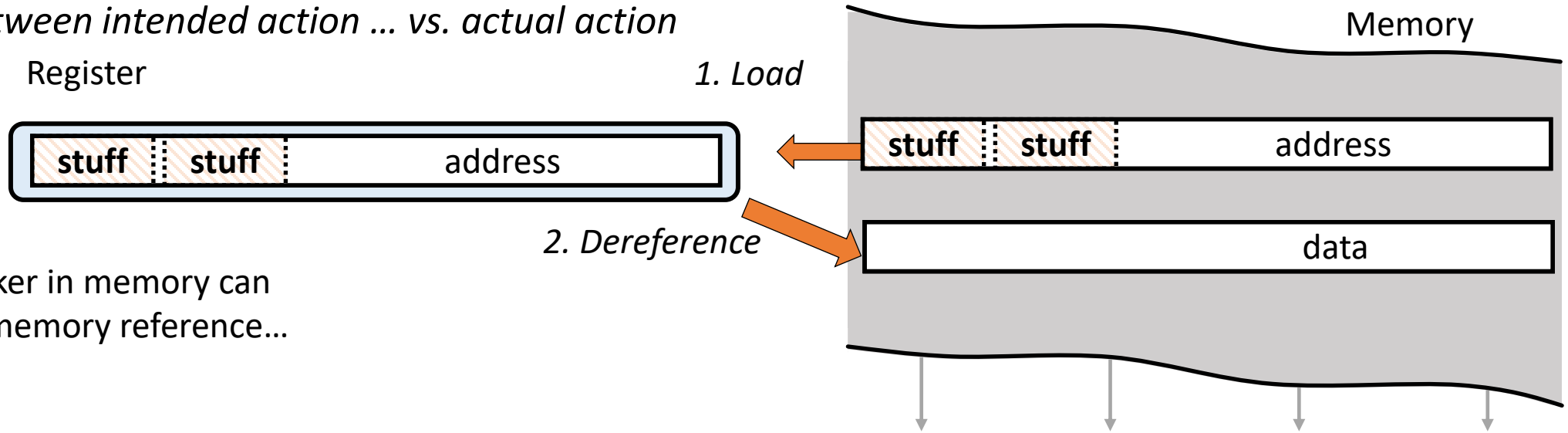
Access controlled by application itself:



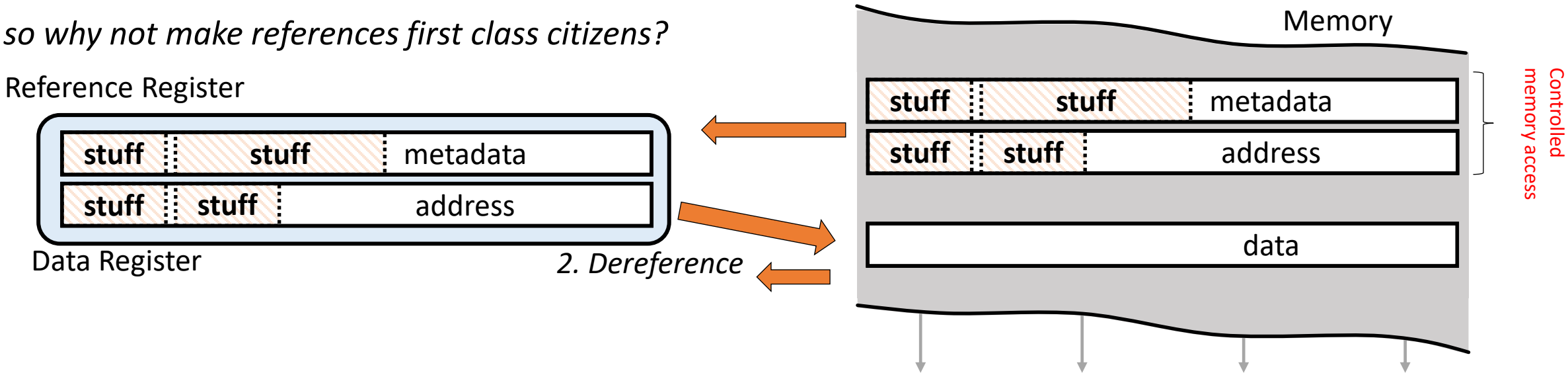
The Case for Hardware Capabilities

- Even with all mechanisms discussed (except for coloring), the memory is still typeless, and dereferencing is unsafe

Mismatch between intended action … vs. actual action



… so why not make references first class citizens?



CHERI is an implementation of a Capability Processor (for MIPS, RISC-V, ARM)

.. From CBG University, now a large U.K / U.S. Government research direction (project Morello, DSbD initiative, ..)

- These mechanisms implement capability protection in hardware. A capability is a resource handle (a pointer) to memory with the property that only if you have the pointer, you can access the memory addressed by it (in most machines a pointer can be constructed at will)
- The idea is not new, but except for some computers in the 1970s the need for memory protection was largely satisified by virtual memory (providing course-grained isolation) since the 1990s.
- VMM is not sufficient any more, memory attacks (ROP, JOP, ..) constitute 70% of all attacks in 2020, 2021

Cambridge CAP computer 1970-79

- Experimental computer designed for ´protection´
  - Pioneered the capability design

https://www.microsoft.com/en-us/research/publication/the-cambridge-cap-computer-and-its-operating-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F72418%2Fcap.pdf

Plessey 250 1972-

- First deployed capability computer (British Army)
  - Used e.g. in Iraq war, 1991 as comm. hub

https://en.wikipedia.org/wiki/Plessey_System_250

IBM System/38 1979-

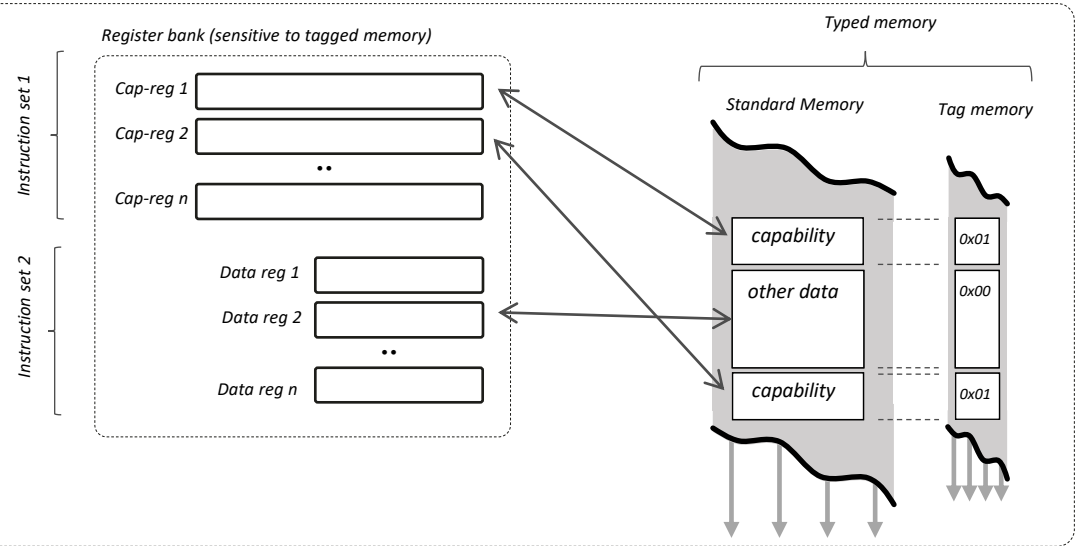
- Maybe the only mass-produced computer to date with capabilities (20.000 sold)

http://bitsavers.org/pdf/ibm/system38/IBM_System_38_Technical_Developments_Dec1978.pdf

Capability Principle (Hardware) 1/2

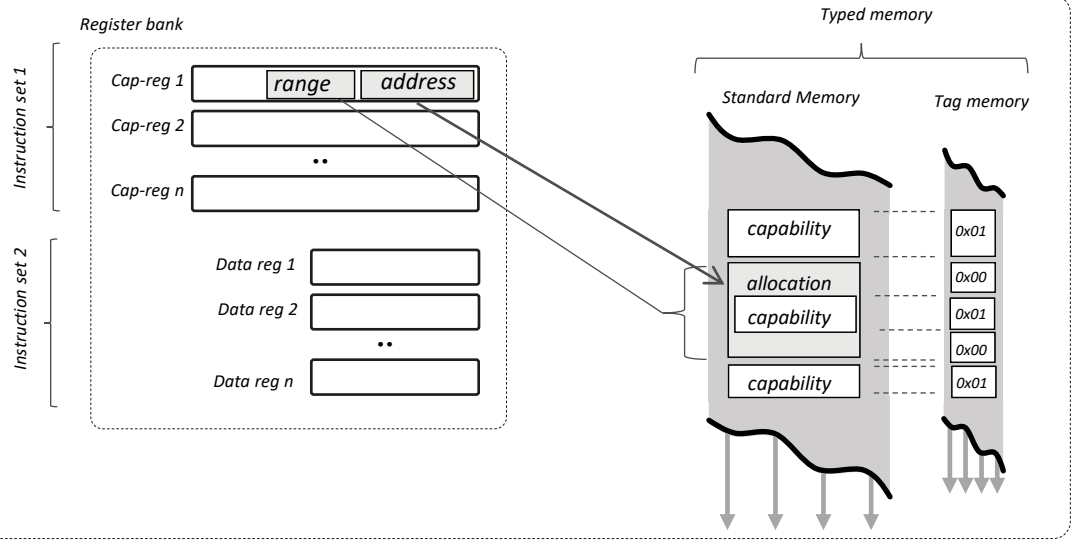
1. Absolute Type Enforcement

- Memory has type. Every memory cell ´knows´ whether it contains a capability or other data
- Capabilities go to capability registers, all other data to other processor registers. I.e. capabilities and data are unconditionally kept separate. If a capability is is loaded from memory as data, its status as a capability is lost.
- Capabilites are modified with instructions separate from data manipulation. I.e. code cannot change capability values in registers outside policy (discussed later)



*2. Capability == **Scope** + policy*

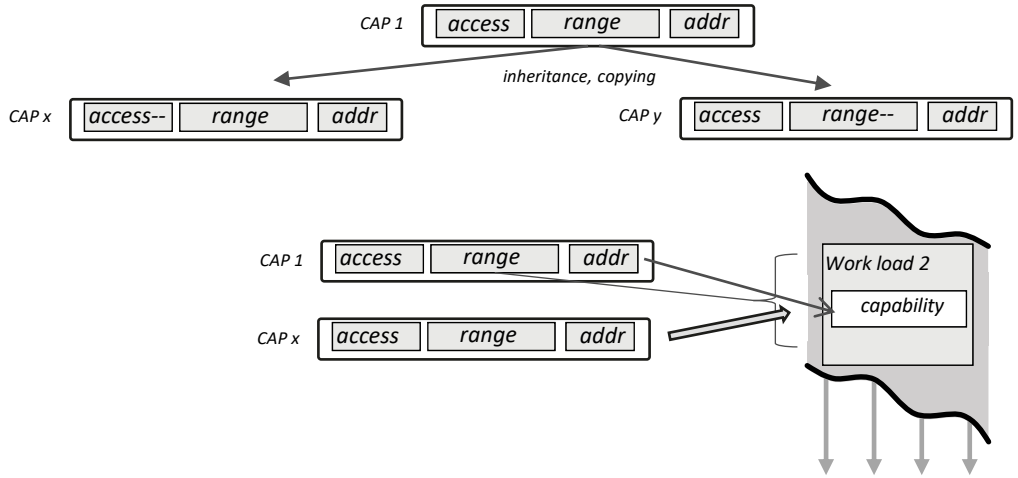
- A memory capability has scope. I.e. In the addition to the reference (pointer), a capability knows the memory boundaries it is allowed to de-reference (read / write) or in case of code - execute
- Note that capabilities are by nature and design recursive. I.e. if an allocation of capability includes other capabilities then these capabilities are loadable by the workload owning the first capability.
- An allocation containing only capabilities is often called a capability table, i.e. that table works as a memory access policy for the workload (function, program ...)



Capability Principle (Hardware) 2/2

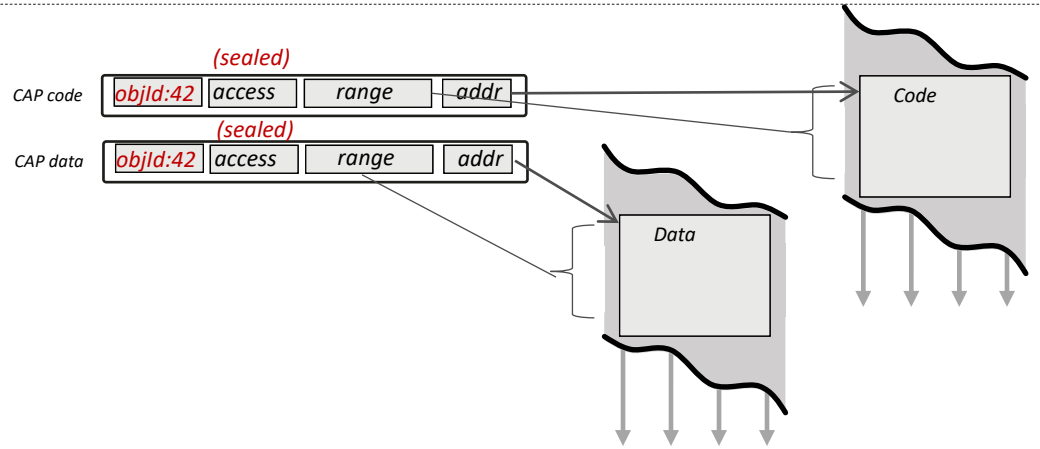
*3. Capability == Scope + **Policy***

- Capabilites can be **copied and modified**. But machine instructions allow only modification that is limiting the scope of the capability
- Capabilities can be **delegated** between workloads. If a workload owns a capability and has access to memory of another workload, it can share its capabilities by writing out the capability to that memory.
- Capabilities in memory can be **sealed** (rendered unusable) and later they can be unsealed (re-activated). One way to do so is by the invocation mechanism described below.



4. Initialization / Invocation / Security Domains

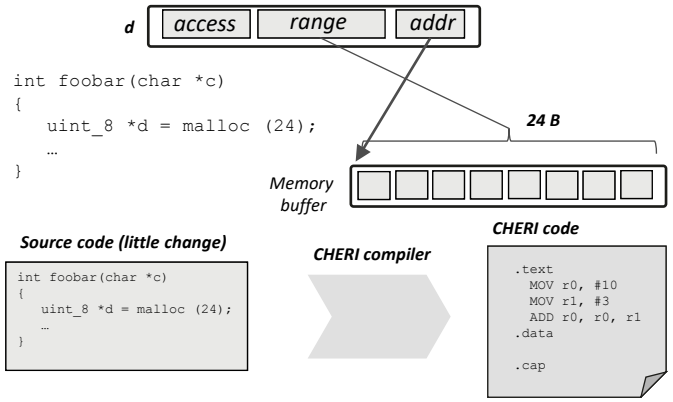
- Two sealed capabilities can be activated with the instruction CInvoke – after the call one ends up being the program counter capability, and the other the data capability
- The two sealed capabilities must have the same object identifier as part of their attributes. This means that the identifier serves as a call gate for the function being invoked – the caller need not have access to either the code or the data of the called function



Hardware Capability Use Cases

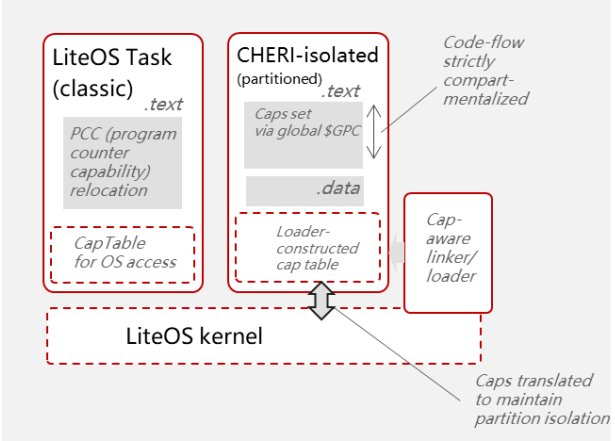
1. Memory Protection

- When CHERI is applied to compiler-generated alloca:s and code-internal memory allocation all **buffers are protected virtually to 100%** due to range protection
- As a result, not only data protection ensues, but **also call-flow will be protected** as no memory violations can take place.
- However, CHERI compilation **does not protect against temporal attacks** (use-after-free). The CHERI team has academic solution for this also, but the solution is fairly high overhead and requires hardware changes



2. Security Domains / Compartmentalization

- In mid- to high end processors, compartmentalization is effectively achieved using virtual memory (MMU) with n stages.
- For small controllers without virtual memory support, CHERI (and the concept of ´cInvoke´) provides an **effective way to isolate workloads** co-existing in the same memory space.
- For kernel compartmentalization (kernel modules, **privilege kernels**) the same applies. CHERI can be used to achieve strong memory separation, therefore significantly reduced attack surface.



Adding support for Hardware (in Compiler)

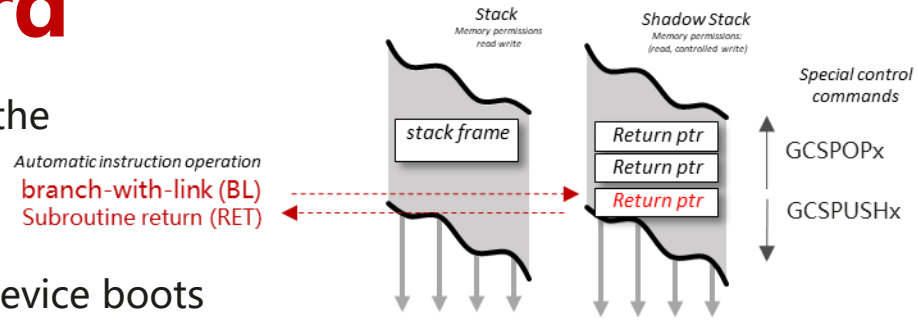
Basic protections are straight-forward

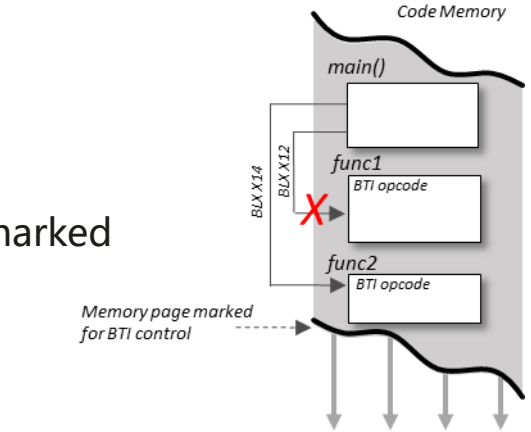
- If the hardware primitive is designated for a specific purpose, its implementation in the compiler is not a big deal.

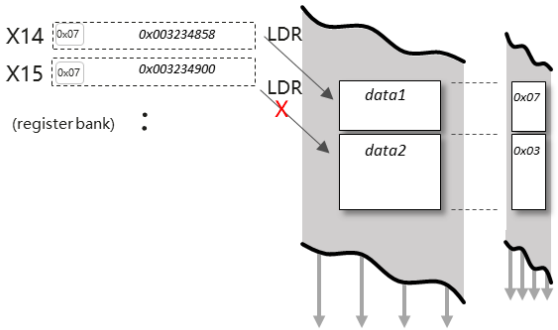
- Consider a Shadow Stack (Control – Flow):
  - **Enable the function** in hardware. This likely means setting up registers when device boots
  - Support the feature **(a) in kernel**, and **(b) in system loader**.
  - Compiler adds special code(?) in function entry (and possible validation in function return)
  - Handle stack unwinding in exceptions (logjump / setjmp)

- Or Branch target protection (BTI)
  - **Enable the function** in hardware. This likely means setting up registers when device boots
  - Support the feature **(a) in kernel** for and **(b) in system loader** - memory pages need to be marked
  - Compiler adds special **magic markers** (pseudo-instructions) before every function entry, and uses **designated branch instructions** (not in ARM) to indirectly jump to functions

- Or basic heap protection with MTE
  - **Enable the function** in hardware. Mark memory with kernel and loader
  - Make designated MTE heap allocator (malloc)
  - No compiler addition: Every time you malloc, assign random color to pointer, and memory. If there are overflows to memory of different color, hardware will trap







But is this all? Can we do more with these primitives? Can safety be further improved?

-- three examples will follow where Aalto SSG researchers have contributed to the state-of-art (only PA)

Pointer Integrity: Memory Safety for Pointers

PI ensures pointers in memory remain unchanged

- Code pointer integrity implies CFI
  - Control-flow attacks manipulate code pointers
- Data pointer integrity
  - Reduces data-only attack surface
  - Prevents all known Data-Oriented Programming (DOP) attacks

PA prevents arbitrary pointer injection

- PAC computation based on key, modifier and pointer
- Modifiers do not need to be confidential
  - Visible or inferable from the code section / binary
- Keys are protected by hardware and set by kernel
  - Attacker cannot generate PACs

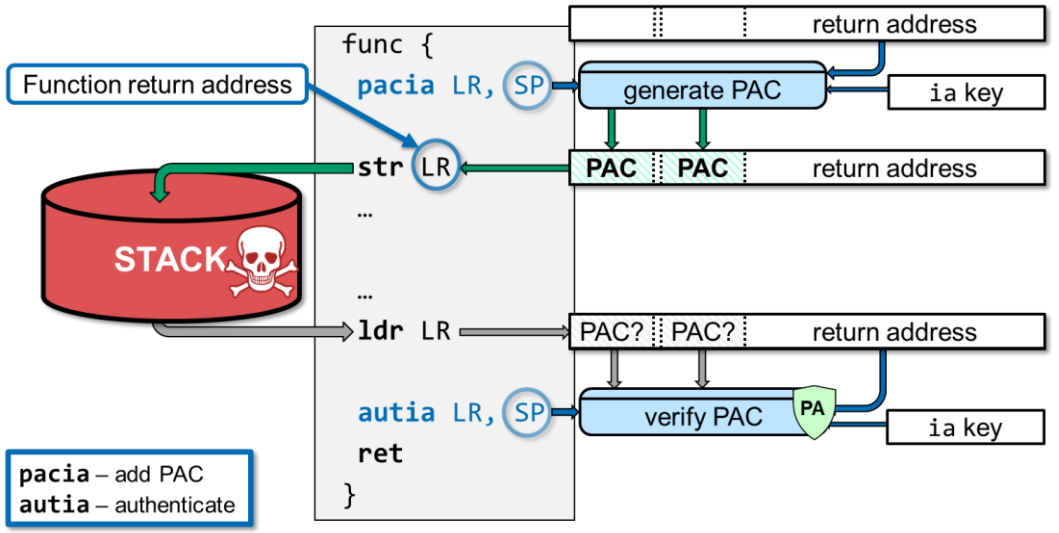
On choosing the PAC modifier:

Without modifier all signed pointers are interchangeable

- Unique for each pointer and pointer value
  - must be available at both creation and authentication
  - must not be modifiable by attacker
- But
  - Static modifiers cannot be unique for conditionally assigned pointers
  - Using a nonce as a modifier needs to be stored securely

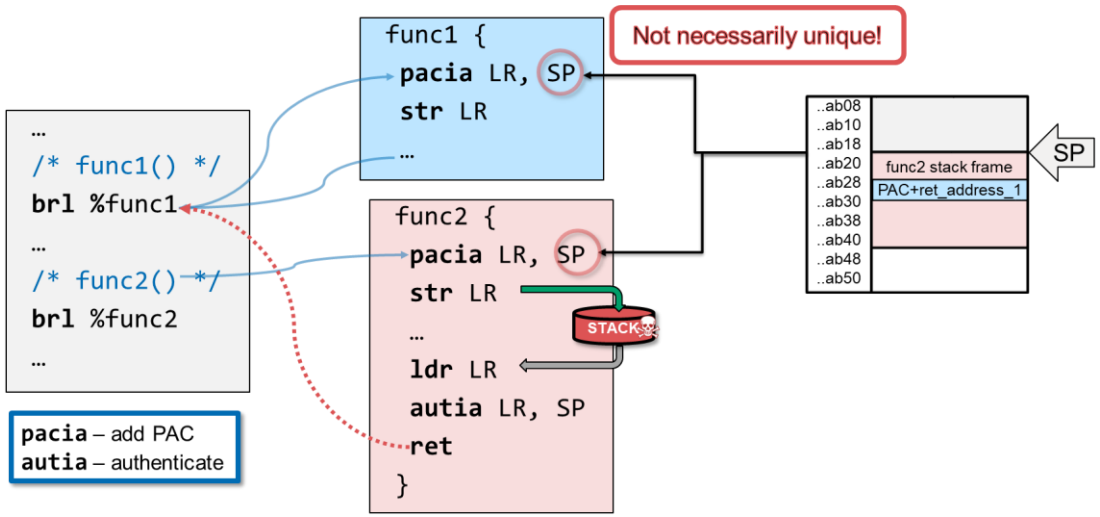
Example: PA-based return address signing

Deployed as **-msign-return-address** in GCC and LLVM/Clang



PA only approximates PI

Adversary may e.g. **reuse** PAC



PA-assisted Run-time Safety (PARTs)

Expands scope of PA protection
- Return address signing
- Code pointer signing
- Data pointer signing

Mitigates pointer reuse by binding
- return addresses to the function definition
- code and data pointers to the pointer type

PARTs Data-Pointer signing

Modifier: **Pointer Type**
- Type assigned at compile time

On-load authentication:
- Improves performance, e.g. only one authentication when iterating arrays
- Register allocation causes a challenge
  e.g., how to handle register spills securely?

PARTs Return-Address signing

Modifier: **SP + function-id**
- ID assigned at compile-time
- Prevent cross-function reuse

```
func {
  mov Xmod, SP
  mov Xmod, #f_id, #lsl_16
  pacia LR, Xmod
  …
  mov Xmod, SP
  mov Xmod, #f_id, #lsl_16
  retab Xmod
}
```

PARTs Code-Pointer signing

Modifier: **Pointer Type**
- Type assigned at compile time

On-use authentication:
- Branches use combined auth+branch instr. (**lbraa**)
- No intermediate authentication

```
// void (*Xptr)(void) =
  …
  mov Xmod, #type_id
  pacia Xptr, Xmod
```

PACed only on pointer creation!

```
// Xptr();
  …
  mov Xmod, #type_id
  lbraa Xptr, Xmod
  …
```

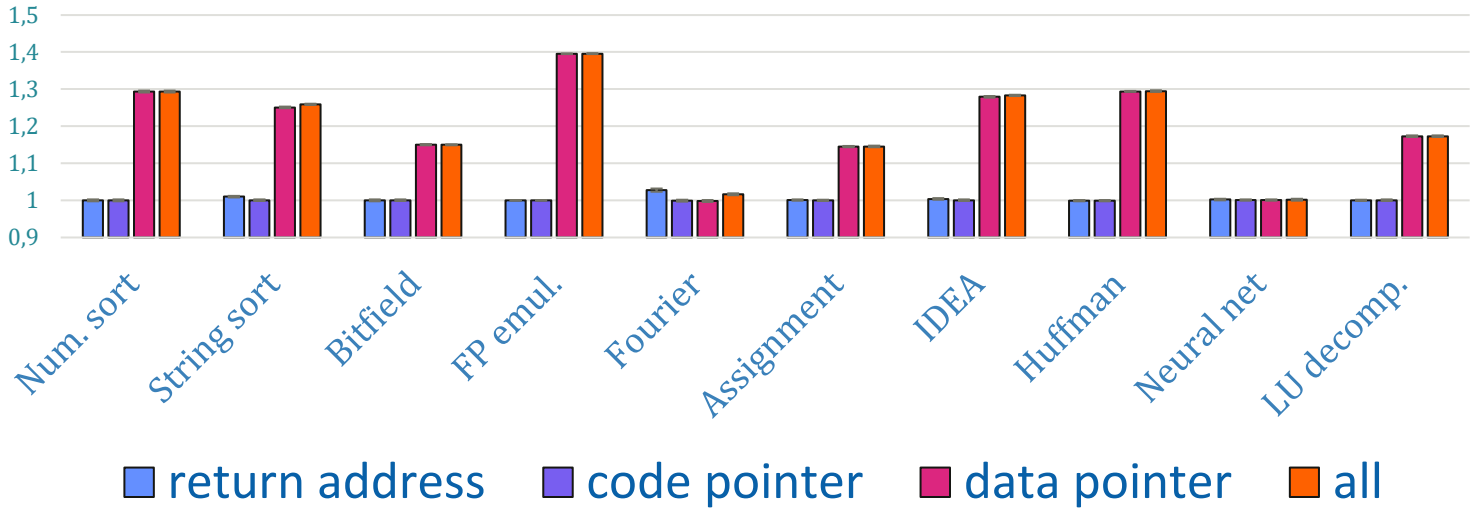
Authenticated on use

pacia – add PAC with instr A-key
lbraa – authenticate and branch

PA-assisted Run-time Safety (PARTs)

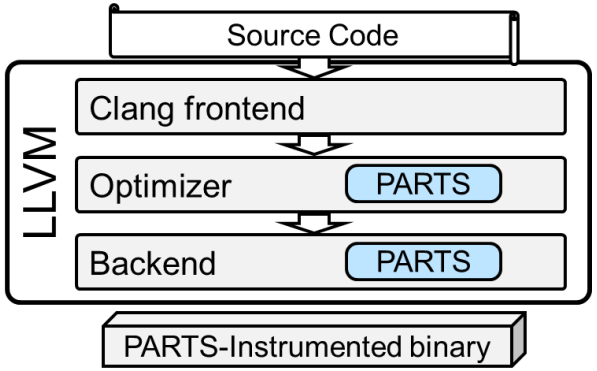
Nbench benchmarks (later SPEC 2017)

- Functional evaluation on ARM FVP simulator for correctness
- Estimated performance overhead based on 4-cycles per PA instruction
  - Return address signing < 0.5% (geo.mean)
  - Code pointer signing < 0.5% (geo.mean)
  - Data pointer signing ~19.5% (geo.mean)



LLVM -based instrumentation

- Using `optimizer` for high-level instrumentation
  - Using LLVM intrinsics for pointer type handling
- AArch64 backend modifications
  - Lower intrinsics to HW-specific instructions
  - Recognizing and protecting register spills



Stack Canaries

- A cost-effective low-overhead protection
  - Use canary value, which is corrupted on overflow
  - Does not prevent overflow, but allows detection of overflows before return
- Widely deployed and supported by compilers
- Canary set at function entry, verified before function return
- Reference value (as a rule) generated at startup

Circumvention:

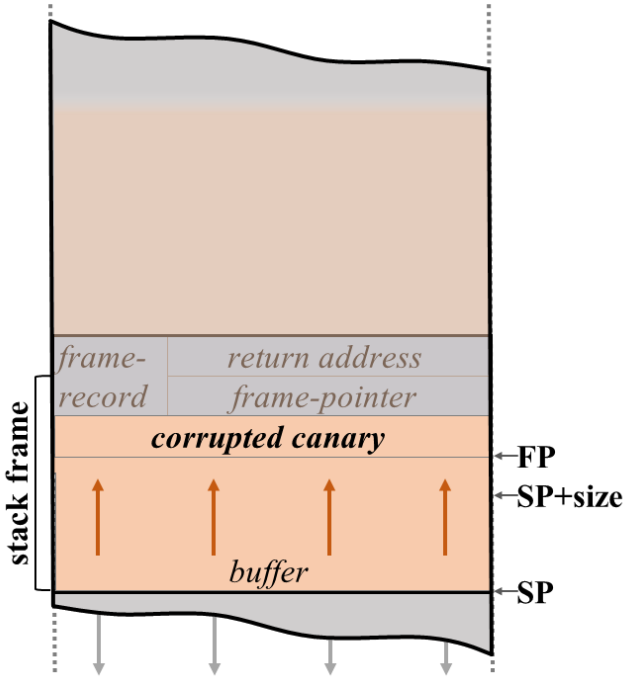
Avoiding the canary
- Arbitrary writes can always be used to avoid canaries
- Overflows can corrupt stack while avoiding canaries

Writing back the canary on sequential overflow
- Canary location available through binary analysis
- Canary value can be leaked via reference canary or stack
- Guessing the canary value

Requirements:

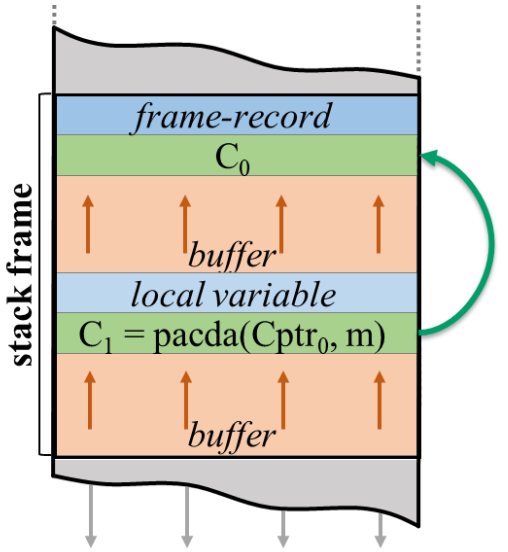
R1: Each canary value should be statistically unique
R2: Reference shall not be available to attacker
R3: Buffer overflow shall corrupt a canary

```
function:
    load canary ← reference_canary
    store canary → stack

    … function body …

    load expected ← reference_canary
    load canary ← stack
    if (expected ≠ canary)
        stack_check_fail()
    return
```



PCan -- Canaries using Pointer Authentication

- PCan uses PA to generate canaries bound to the stack-frame
  (R1: Canary value should be statistically unique)
- PCan relies only on hardware-protected PA-keys, not reference canaries
  (R2: Reference canaries must not be accessible to attacker)
- PCan places fine-grained canaries after each individual stack buffer
  (R3: A stack buffer overflow must always corrupt a canary)



- Implemented on LLVM 8.0
  - Analysis on LLVM IR in optimizer
  - Instrumentation in AArch64 backend
- SPEC CPU 2017
  - Using 96board Kirin 620 HiKey
  - PA-analogue emulates PA overhead*
    - Estimated 4-cycle overhead
    - Memory dependency as expected PA behavior

Masking: return address XOR canary

Benchmark	stack-protector		PCan	
	rel. diff.	std. err.	rel. diff.	std. err.
505.mcf_r	−4.78%	(4.55)	−0.05%	(0.13)
519.lbm_r	−0.01%	(0.01)	0.04%	(0.02)
525.x264_r	−0.01%	(0.01)	1.80%	(0.01)
538.imagick_r	−0.01%	(0.01)	0.19%	(0.01)
544.nab_r	0.05%	(0.24)	−0.18%	(0.16)
557.xz_r	0.00%	(0.03)	0.04%	(0.06)
geo. mean.	−0.08%		0.03%	

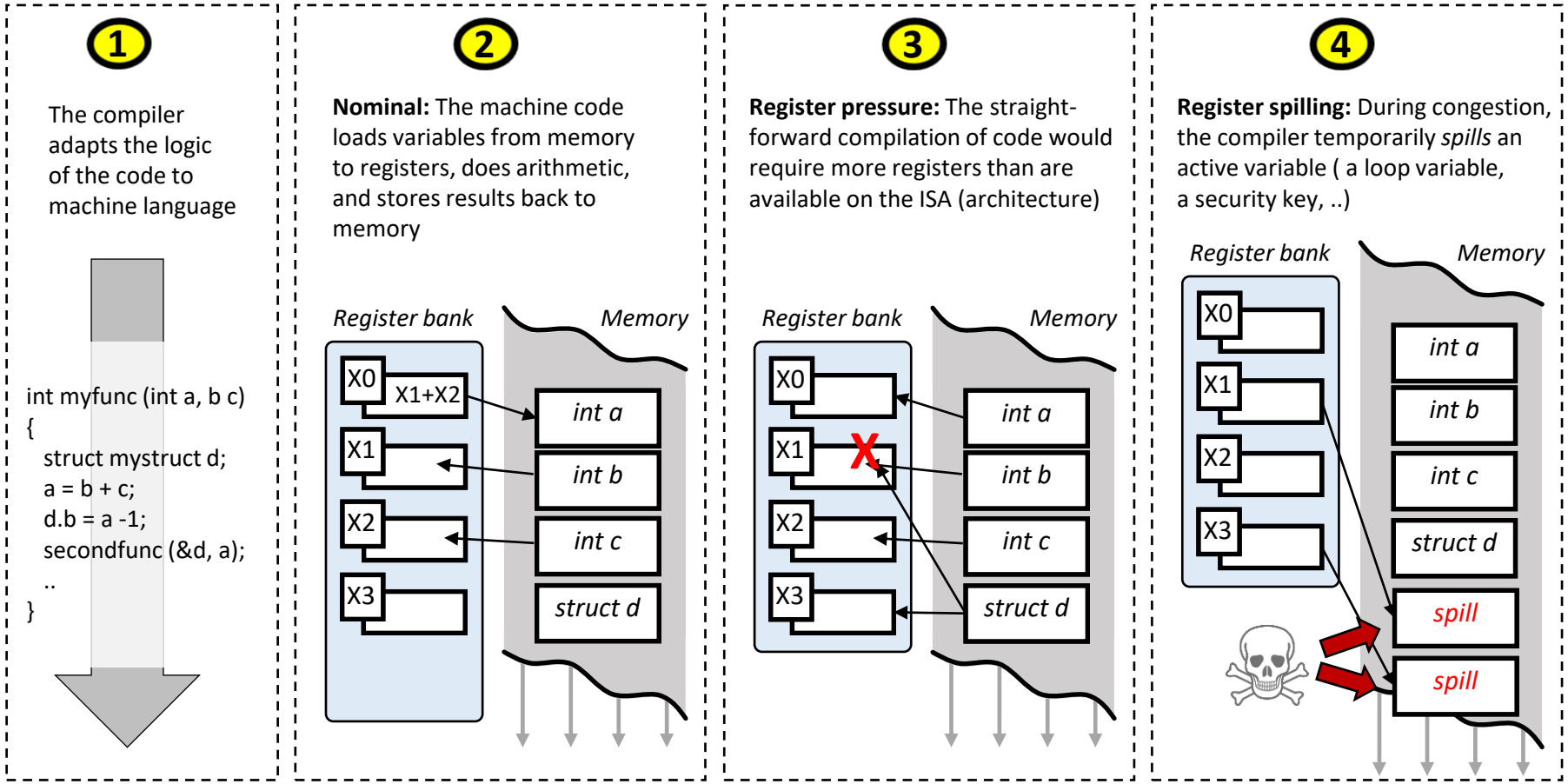
```
prologue:
    mov   x8,  sp              ; x8 ← SP
    movk  x8,  #3,  lsl #48    ; x8 ← m = function_id ‖ SP
    pacga x10, sp,  x8         ; x10 ← C0
    sub   x9,  x29, #0 x8      ; x9 ← Cptr0
    pacda x9,  x10             ; x9 ← C1
    str   x9,  [sp , #40]      ; store C1 → stack
    stur  x10, [x29, #-8]      ; store C0 → stack

epilogue:
    mov   x8,  sp              ; x8 ← SP
    movk  x8,  #3,  lsl #48    ; x8 ← m = function_id ‖ SP
    ldr   x9,  [sp, #40]       ; load x9 ← stack
    autda x9,  x8              ; x9 ← Cptr0 if PAC verified
    ldr   x9,  [x9]            ; load x9 ← stack
    pacga x10, sp,  x8         ; x10 ← C0
    cmp   x9,  x10             ; verify x9 = C0
    b.ne  <fail_check>         ; or fail
```

Register Spilling Protection

(if time permits)

Register Spilling – what is it?

- ✓ The activity of adapting to the limited amount of (general purpose) registers in an architecture when / where needed
- ✓ Security relevant spilling happens frequently: E.g. hidden canaries end up in memory tens of times of in Linux programs compiled with stack canaries: https://pdfs.semanticscholar.org/be68/aa84958398e96331a11aa2e81cb5a3047629.pdf



- ★ Spilling is deterministic (inserted by the compiler), so also statistical (brute-force) attacks can be mounted
- ★ The spilling is NOT visible to the programmer, or even to the compiler front-end.
- ★ A large cause of spilling is the ABI calling convention, i.e. agreement about how to call functions (callee saved registers, caller saved registers)
- ★ For security-specific features (like pointer integrity), e.g. spilling a register that has been validated (but not used) violates the security assumptions of the mechanism. This consideration is present in many published memory protection mechanisms

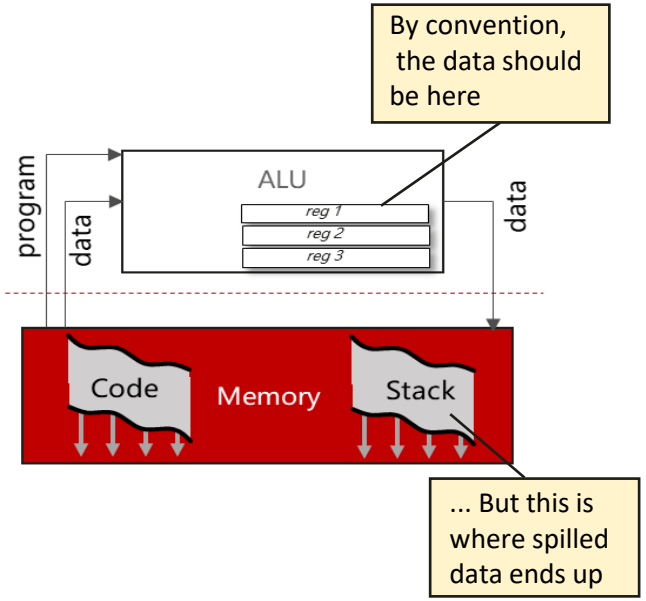
Motivation to even look at Register Spilling for security

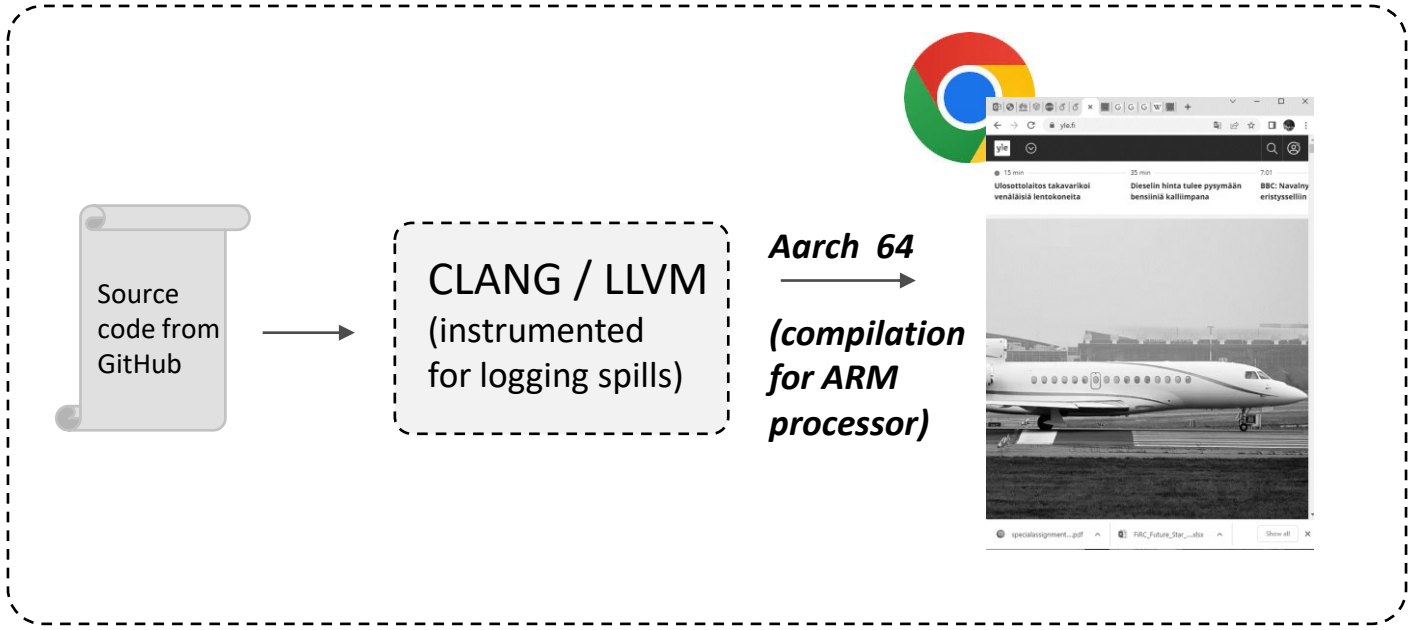
- ✓ Compiler induced register spills are surprisingly commonplace – even in commercial processors with 10+ general purpose registers (more registers, less need for spills)
- ✓ Case in point: When the C/C++ code for the most common browser is compiled, register spills show up in 1.5 million locations in the code

statistic	*value*
nr. of functions	1041705
saved CSRs	3548372
spills inserted	133963
spill slots allocated	112163
reloads inserted	240375
emergency spilled registers	1

Table 1: Overall statistics of compiled code of chromium

- ✓ I.e. programmer's data ends up on the stack without his knowledge and prioritization. This data may be a critical loop variable, part of a crypto-graphic key or a password input

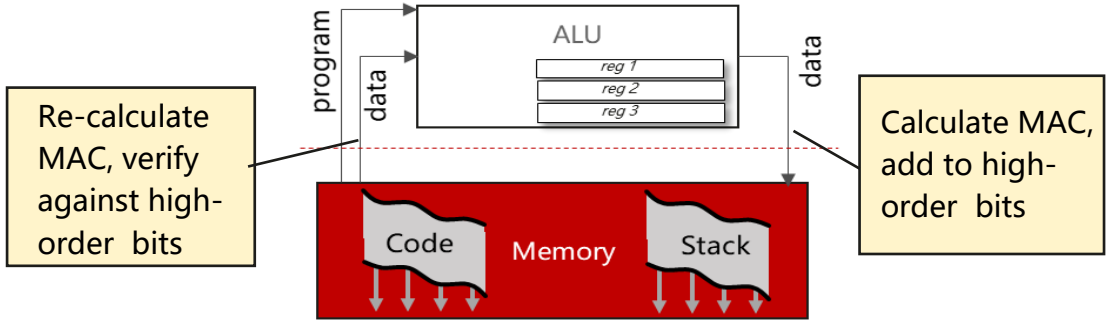




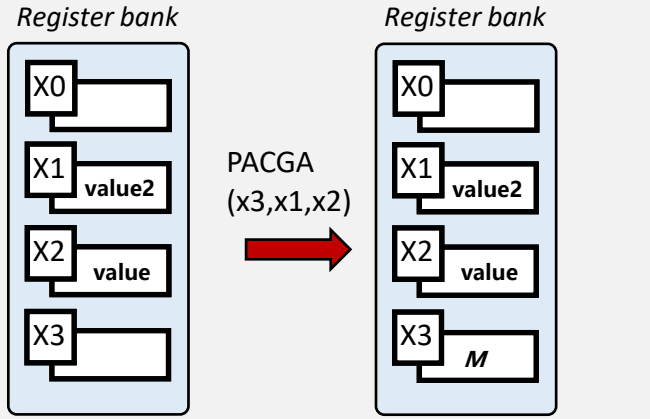
Andrea Fanti, special assignment course, 2020

ARM Pointer Authentication

- ✓ **New architecture and instructions:**
  1) When pointer is moved from register to memory compiler can add code to compute a MAC over pointer
  2) When pointer is retrieved from memory, MAC can be recomputed and verify against code produced at store
- ✓ **Penalty around 4 cycles per operation**
- ✓ **Keys are maintained in hardware (´set by kernel´)**

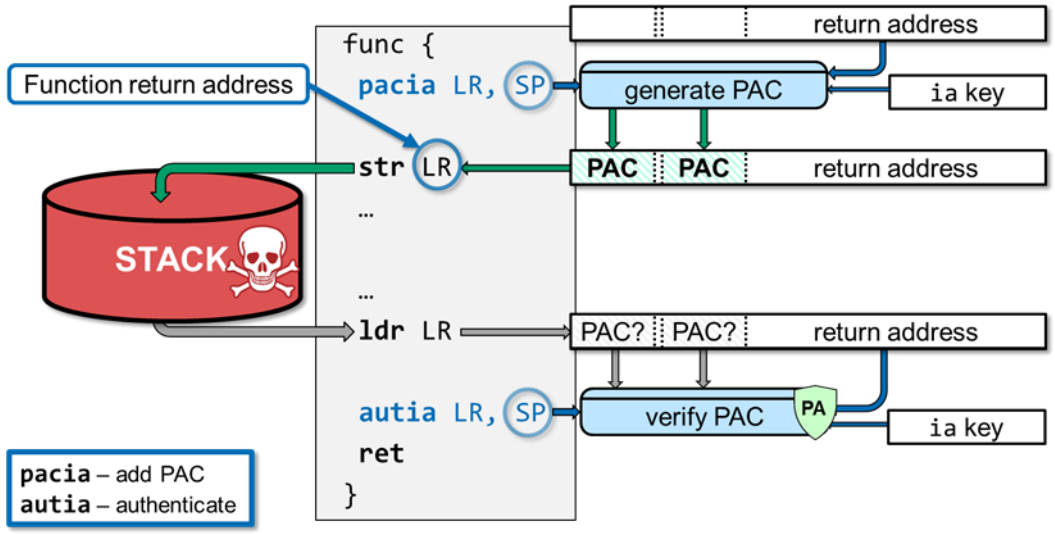


For the work on Register spilling we use a ´general´ variant of the PA primitive: PACGA



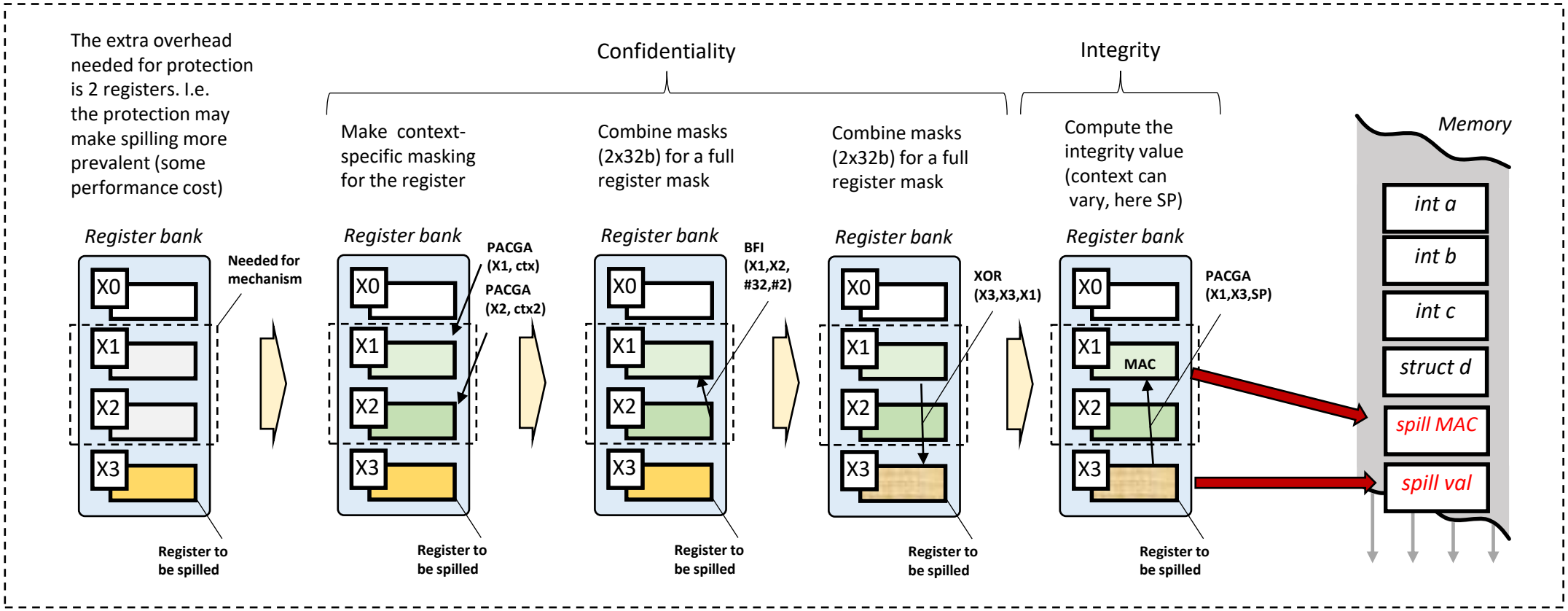
$M=MAC_k(value1,value2)$

Nominal use case: protect function return address



‘Fixing Spilling with PA’ principle

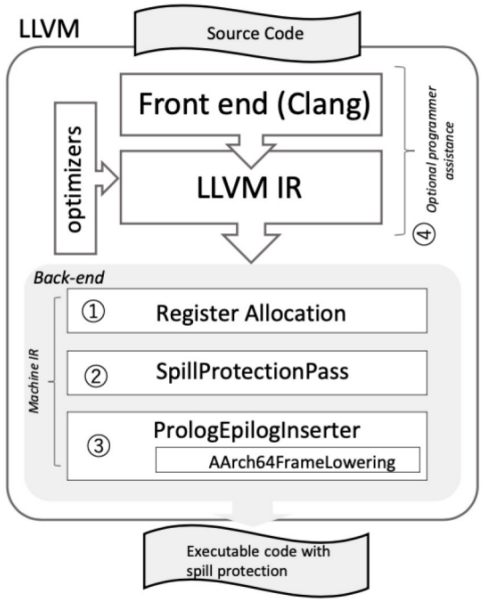
- ✓ Principle: Authenticate and Mask (encrypt) all register spills using Pointer Authentication during compilation
- ✓ One topic is how to optimize, but main mechanism outline can be described as follows:



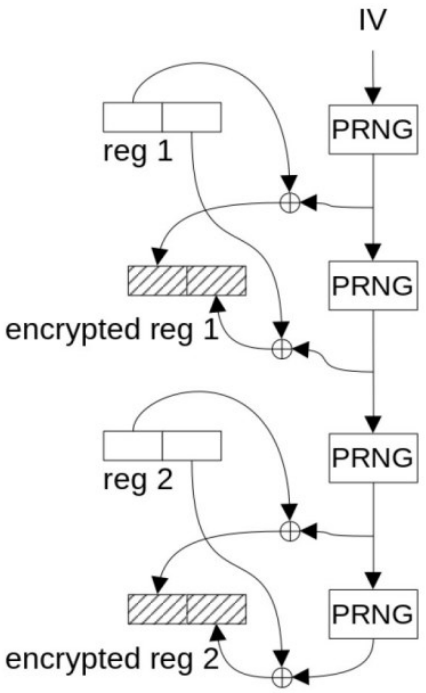
Pending exact details (to be implemented) for spill (and reload) we are looking at around 2 x 10 instructions, 6 PAC instructions, so the cost of a ”protected” spill is in the range of 40-50 cycles. Spills can be categorized into spills needing confidentiality, spills needing authenticity, and spills irrelevant for security

Towards Register Spilling Security using LLVM and ARM PA

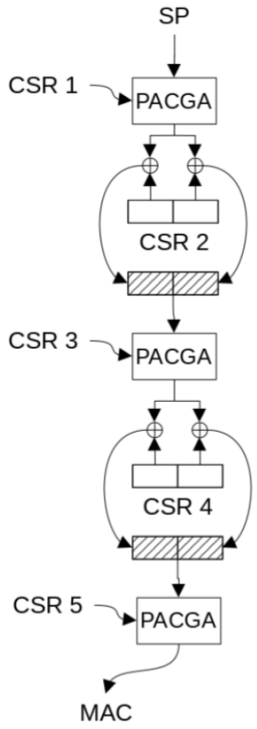
- Using ARM PA to provide confidentiality and integrity for register spills in LLVM
- By using ARM PA ´innovatively´, only 2 registers need to be reserved for this ´system test´



Compiler was augmented in optimizer / backend phases



Keystream generation for confidentiality



´Packing callee-saved registers into a common MAC'

```
// Mv  := original spill slot stack offset
// xv  := register whose value to spill
// d   := nonce to use, d = hash(function || Mm)

    // Storing registers
    mov     x15, #d
    pacga   x15, x15, sp
    eor     x14, xv, x15     // encrypt left word
    pacga   x15, x15, sp
    eor     x14, x14, x15, lsr #32
                             // encrypt right word
    str     x14, [sp, #Mv]   // store encrypted value
    pacga   x15, x14, sp     // generate MAC
    lsr     x15, x15, #32
    str     w15, [sp, #Mm]   // store MAC
    ...
    // Loading registers
    ldr     xv, [sp, #Mv]
    pacga   x15, xv, sp      // regenerate MAC
    ldr     w14, [sp, #Mm]   // reload MAC
    lsl     x14, x14, #32
    eor     x15, x14, x15    // compare MACs
    cbnz    x15, .Lfail      // fail if corrupted
    mov     x15, #d
    pacga   x15, x15, sp
    eor     xv, xv, x15      // decrypt left word
    pacga   x15, x15, sp
    eor     xv, xv, x15, lsr #32
                             // decrypt right word
```

Annotated example of combined integrity and confidentiality protection

Register Spilling – performance (does it really matter when done in isolation?)

- Binary overhead is around 12% for integrity and around 35% for confidentiality
- Tests done on an Apple M1 core with Linux, i.e. realistic tests on real hardware
- We also confirm that the expected 4 cycles per PA operation is quite accurate in real HW
- The obvious next step is to add programmer assistance to indicate which buffers and variables are to be considered important enough to protect

Protecting all spills is prohibitively expensive

TABLE II: Spill protection execution time overhead measured on SPEC CPU 2017 benchmarks.

Benchmark	Baseline	With integrity	Overhead	With encryption	Overhead
perlbench	62.39 s	125.96 s	101.89%	397.83 s	537.64%
gcc	31.02 s	46.89 s	51.18%	120.64 s	288.92%
mcf	231.70 s	242.06 s	4.47%	306.44 s	32.26%
omnetpp	267.62 s	305.57 s	14.18%	477.70 s	78.50%
xalancbmk	209.34 s	230.60 s	10.16%	329.35 s	57.33%
x264	62.88 s	81.49 s	29.59%	197.29 s	213.75%
deepsjeng	249.76 s	285.75 s	14.41%	549.63 s	120.06%
leela	333.92 s	395.62 s	18.48%	695.16 s	108.18%
xz	87.90 s	91.23 s	3.80%	107.54 s	22.34%

TABLE III: Spill protection stack size and spill count overhead measured on SPEC CPU 2017 benchmarks.

Benchmark	Spills (no instr.)	Spills (instr.)	Spills overhead	Stack size (no instr.)	Stack size (instr.)	Stack size overhead
perlbench	16779	16801	0,13%	244176 B	264240 B	8,22%
gcc	79723	79769	0,06%	847904 B	963904 B	13,68%
mcf	326	335	2,76%	67568 B	68080 B	0,76%
omnetpp	27478	27483	0,02%	439408 B	479040 B	9,02%
xalancbmk	58651	58652	0,00%	880512 B	964144 B	9,50%
x264	9327	9583	2,74%	313056 B	326704 B	4,36%
deepsjeng	711	713	0,28%	25488 B	26256 B	3,01%
leela	1780	1789	0,51%	197472 B	199536 B	1,05%
xz	1671	1703	1,92%	72784 B	75584 B	3,85%

Spills w.o. our system and our register reservations

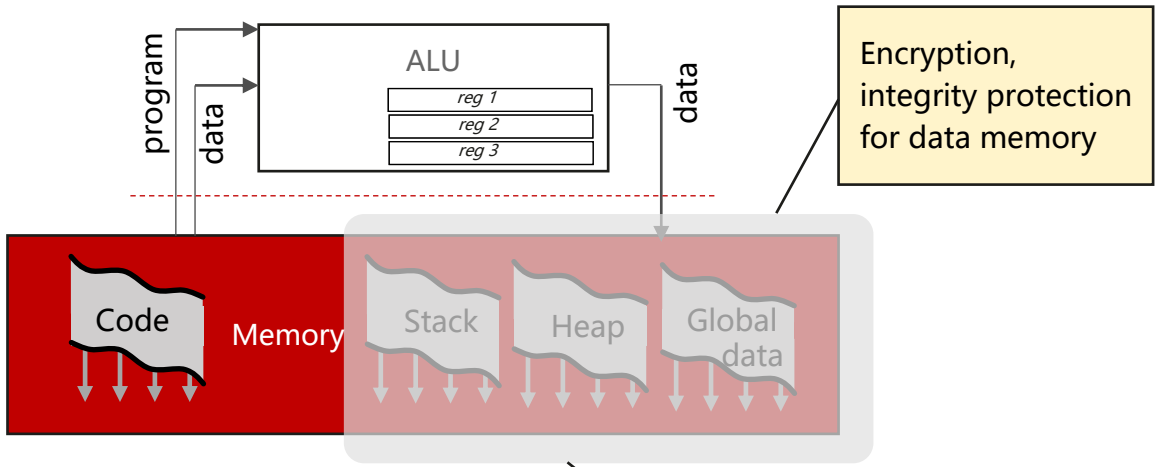
Number of spills in SPEC 2017

An interesting corollary of the results of this paper

- *In this work we fully protected memory storage (for spills), and do not use memory -- only registers – for doing it, using ARM PA*
- *Spilled registers are not objectively different from stack allocations, heap or global data, on instruction level, we are dealing with loads & stores, from memory to 64-bit registers, and arithmetic over registers, nothing more*

→ *So there should be no theoretical or compiler engineering issue to integrity-protect and make confidential every single byte that is ever stored in memory*

(Of course the performance overhead would be disastrous)



Some Thoughts for Future

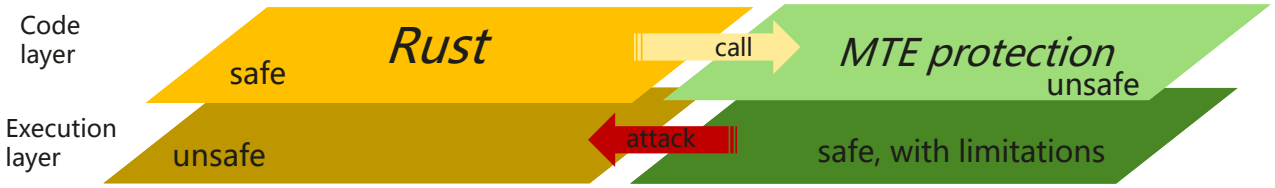
Mixing Language-Based and Hardware-Assisted Protection

- For system programs written in Rust, the code is argued to be memory safe by virtue of the language and the compiler.

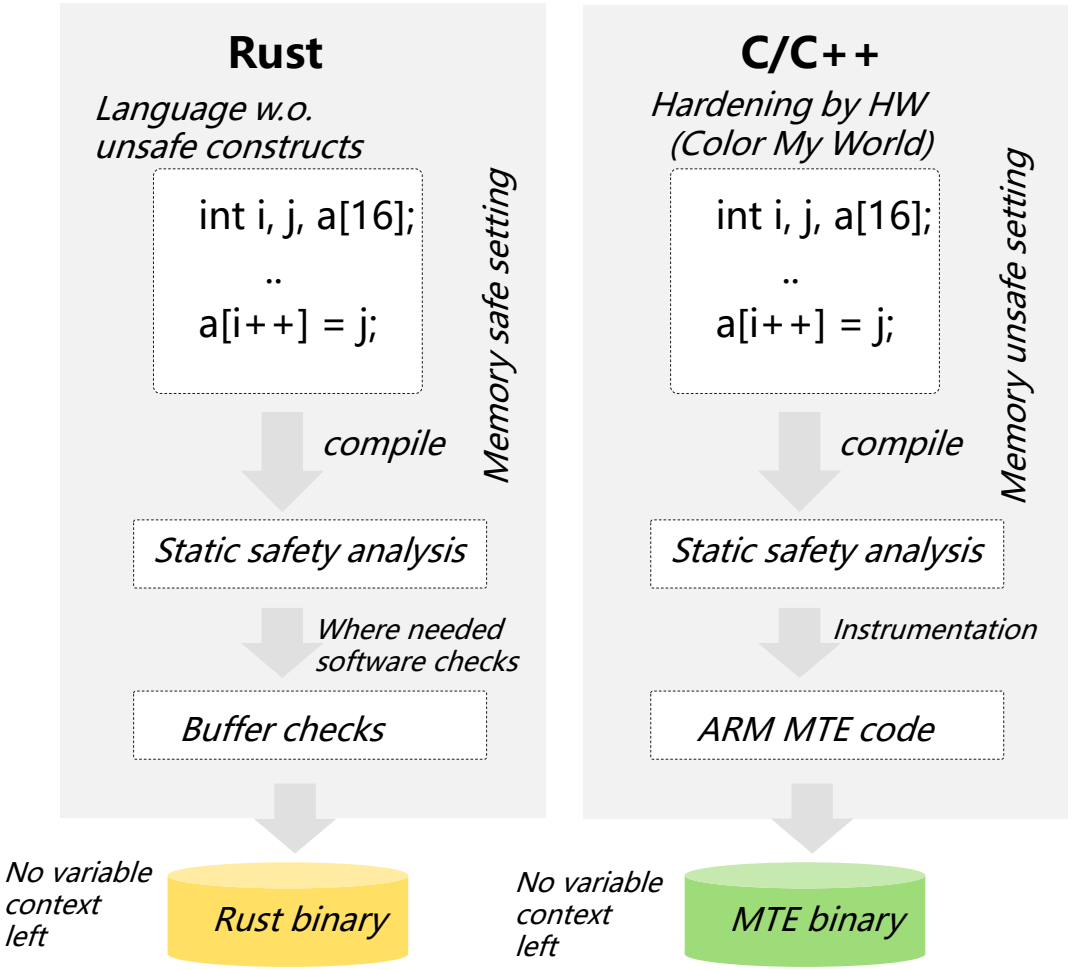
- If low-level (HW) access is needed from Rust (or legacy library interaction), Rust includes an "unsafe" mode / API, where expectation is that the code on the unsafe part is verified (formally?) to be bug free

Wrong assumption!

Just combine the two mechanisms for protection



Example: Stack frame / function return is implicitly safe in Rust. It is explicitly safe in MTE system, but MTE is not guaranteed to be without attack vectors for all allocations



Approach:

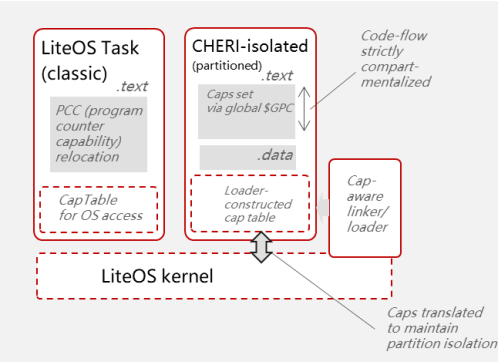
- identify safe and unsafe allocations across whole program (both parts). Rust compiler helps a lot (keeps information)
- use evolution of Steensgard´s *safely points-to* algorithm (1996), especially useful at the Rust/C interface, since allocations may be used cross-wise
- Instrument whole program according to 'Color My world' principle.

https://www.cs.cornell.edu/courses/cs711/2005fa/papers/steensgaard-popl96.pdf

Computer / Computation Security beyond memory protection only (helicopter view)

Hardware Capabilities

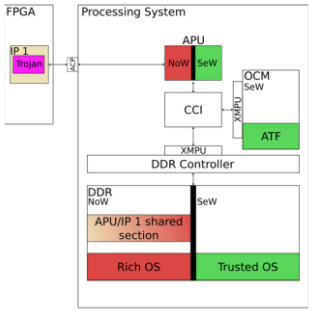
- Has re-entered the research community via The Cambridge University CHERI project
- In essence, fine-grained access control for memory references with very high (byte) resolution
- Can be used for memory protection, compartmentalization (in linear memory), as a system access control feature, distributed access control (macaroons). Biggest unresolved problem in temporal safety.



Programmable Hardware as a Security Enabler (and attack vector)

- **Hardware Trojans:** The more co-designs with CPUs + FPGA with access to the internal bus, the more problems
- **Sustainable Security:** To insert hardware reconfiguration for post-deployment protection (e.g. when new side channel attacks are invented). Project Hydranos, University of Darmstadt
- **MetaSys** (e.g)**:** Research Vehicle for Software-Hardware Securty+performance Co-Design. RISC-V setups for protyping secure architecture in processor pipeline, memory and instruction microcode.

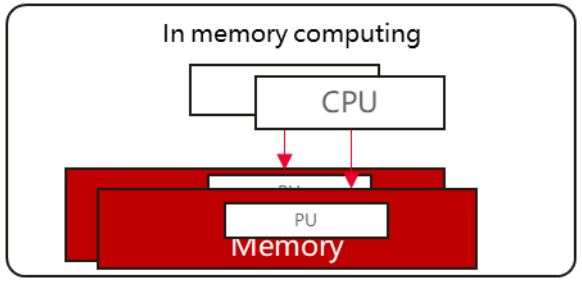
Breaking TrustZone



https://link.springer.com/content/pdf/10.1007/s13389-021-00273-8.pdf

Computer Architecture is changing – security with it (?)

- **In-memory computing:** May replace vonNeumann after Moore´s law finally dies
- **Non-volatile RAM:** May invalidate device reboots as a security / integrity feature
- **Quantum / Neurmorphic computing:** Even the security requirements are unclear.



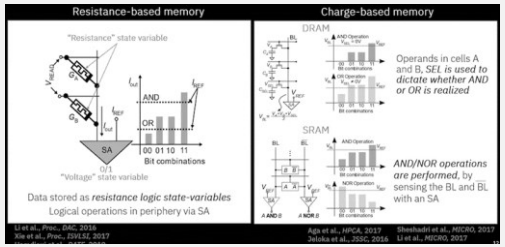
In-Memory Computing Security Panorama

PIM
(processing in memory)

PUM
(processing using memory)

Use physical properties in memory to compute

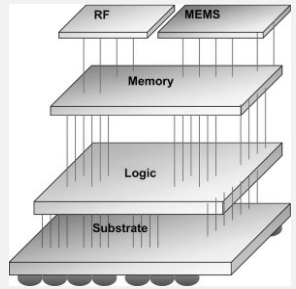
- Analog domain
- Massively parallel
- Further in the future



PNM
(processing near memory)
.. also C-RAM:
Computational RAM

Add (localized) computing elements into the memory chip:

- Existing prototypes available
- Possibly significant security (and safety) concerns
- Big boost to host performance



Attacks

https://dl.acm.org/doi/pdf/10.1145/3386263.3411365

Many attack vectors are completely new:

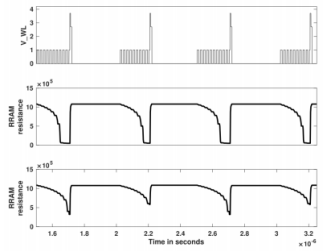
PIM for eavesdropping (steal memory contents)

PIM for corruption (overwrite memory contents)

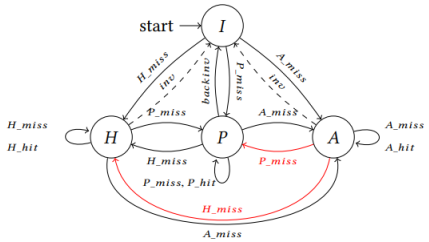
Memory corruption due to faulty PIM code

Non-volatile PIM → offline attacks

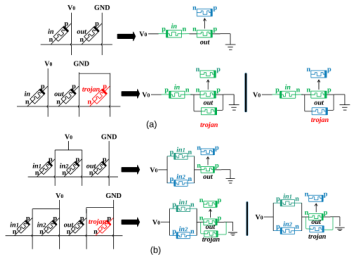
Rowhammer-style wear-out (RRAM)



Misguided cache / PIM interaction
(cache needs to be invalidatable from memory side)
.. confused deputy attack



In-memory hardware trojans



Interaction woes

The software-hardware interaction changes:

- How to program PNM units ?
- How to handle data / code cache coherency ?
- How to handle privilege? types ?

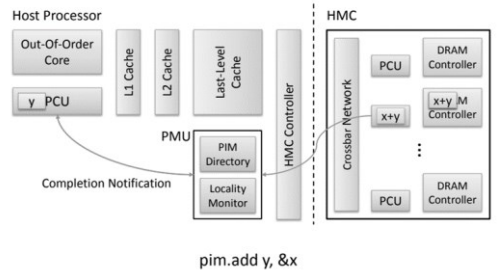
```
for (v: graph.vertices) {
  value = weight * v.rank;
  for (w: v.successors) {
    __pim_add(&w.next_rank, value);
  }
}
pfence();
```
pim.add r1, (r2)
pfence

- How much can a compiler help ?
- How to handle multi-user / multi-workloads (kernel)?

+

- What is integrity if memory can change itself? Isolation?
- Access control? Could actually PIM == memory firewalls?

One architecture proposal



http://users.ece.cmu.edu/~omutlu/pub/pim-enabled-instructons-for-low-overhead-pim_isca15-talk.pdf
http://users.ece.cmu.edu/~omutlu/pub/pim-enabled-instructons-for-low-overhead-pim_isca15.pdf

The End