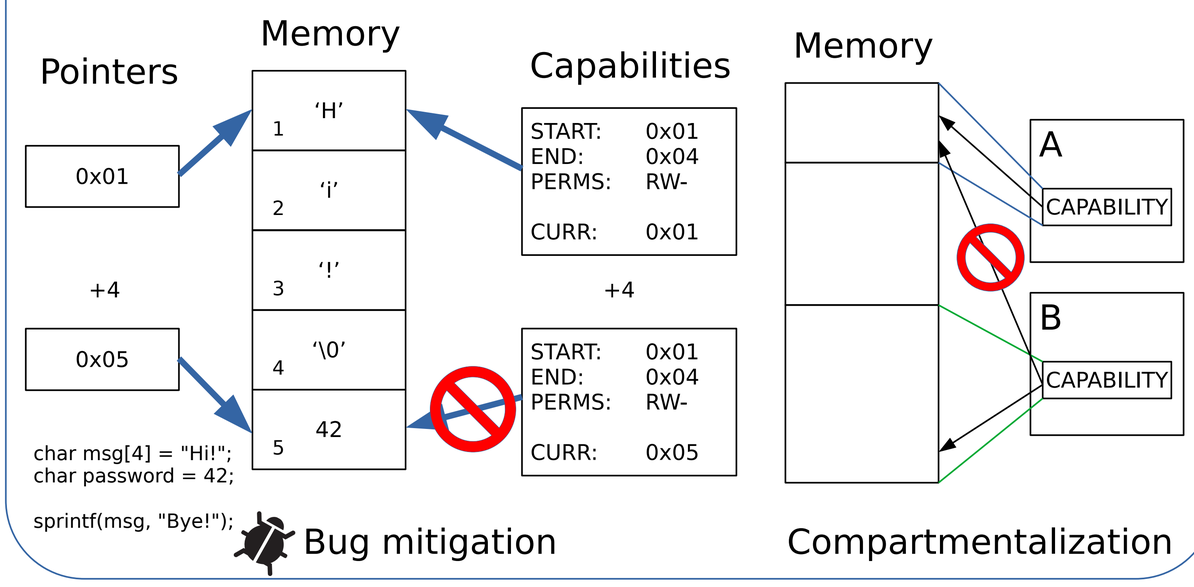
Linear capabilities for CHERI

An exploration of the design space

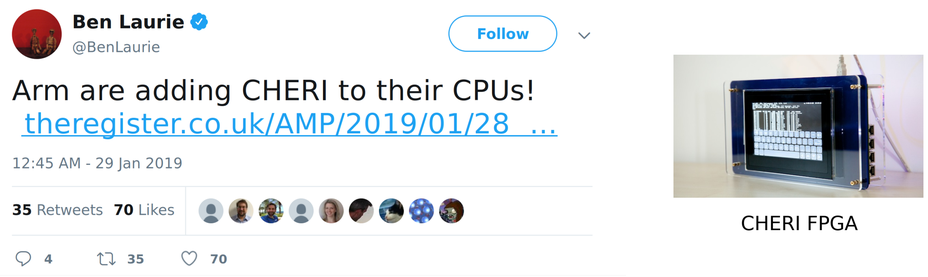
Aaron Joos Lippeveldts Dominique Devriese

What are capabilities?



CHERI

Capability Hardware Enhanced RISC Instructions (**CHERI**) is a research project that explores using **capabilties** through an **instruction set extension** in order to better support **software compartmentalisation**.



CHERI FPGA

www.cl.cam.ac.uk/research/security/ctsrd/cheri/

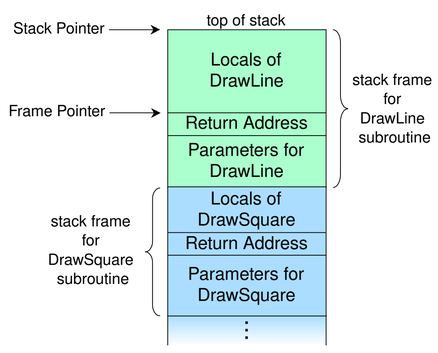
Why linear capabilities?

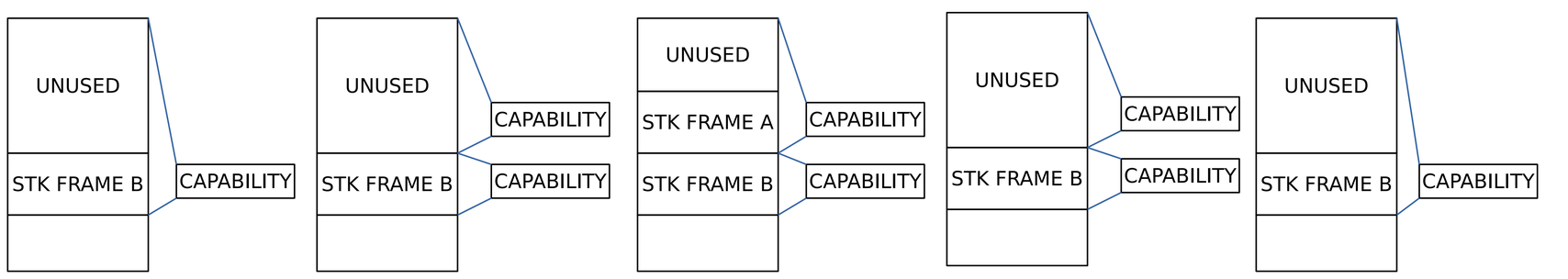
Problem with capabilities
- Hard to revoke given authority (capabilities)

Linear capabilities for fully abstract compilation of separation-logic-verified code
dl.acm.org/citation.cfm?id=3341688

Possible solution
- Linear capabilities (cannot be copied, only moved)

→ Enables **StkTokens**: a secure calling convention
Benefits: *well-bracketed control flow* and *stack frame encapsulation*, even when invoking adversarial code

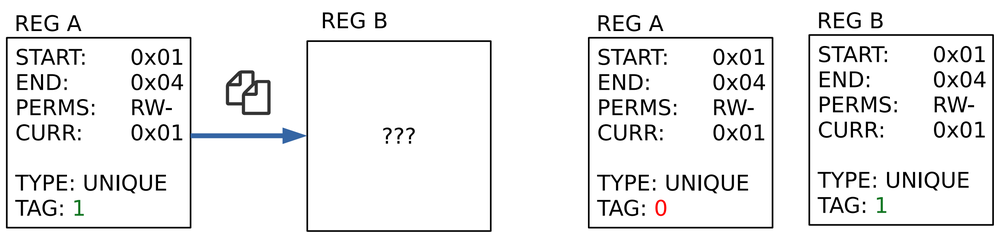




Linear capabilities for CHERI

- Add linearity representation
- Respect linearity in existing instructions
- Add new instructions to deal with linearity

3 types of instructions to be modified
- Capability modification instructions (CandPerm, ...)
- Load/Store instructions (CLC, ...)
- PCC-related instructions (CJR, ...)

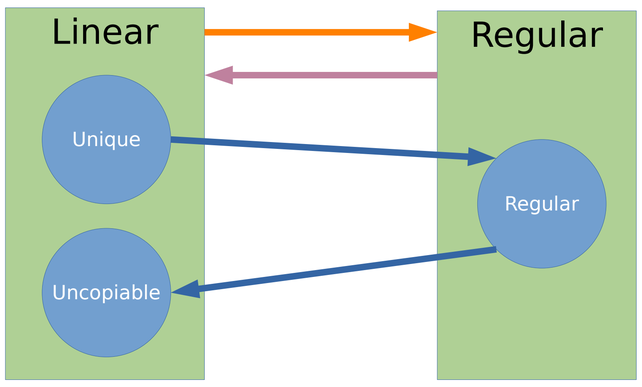


Example pitfall: delay slot on MIPS

Instruction	Description
CSetLinear	Set linearity type (privileged)
CWeakenLinearity	Weaken linearity type
CGetLinear	Get linearity type
CSplit	Split a capability in two
CSplice	Splice two capabilities

Implemented in QEMU

Multiple options for linearity types



Model	Guarantee	Advantages
→	Noncopiable	Easy to deploy
→	Globally unique	Strong global guarantee
→	Depends on type	↑ Both

Evalation
- Basic validation of implementation with small tests
- Program implementing StkTokens (not tested on sim)
  practical challenge: how to obtain seal capabilities?

```
__asm__ volatile (
"fac:"
    "slti $3, $6, 2;"              // set $3 to ($6 <= 1)
    "beqz $3, 20;"                 // if $3 is zero, jump ahead 20 bytes (+5 insns)
    "nop;"                         // branch delay
    "daddi $2, $zero, 1;"          // set 1 as return value
    "cmove $c4, $c26"              // *return our stack ($c26 will be overwritten)
    "ccall $c1, $c2, 1;"           // *return
    "nop;"                         // branch delay
    "daddi $3, $zero, -772;"
    "cincoffset $c26, $c26, $3;"   // make room on stack for 3 caps and 1 int
    "csd $6, $zero, 0($c26);"      // put n on the stack
    "csc $c1, $zero, 32($c26);"    // put the code return cap on the stack
    "csc $c2, $zero, 288($c26);"   // put the data return cap on the stack
    "cgetoffset $3, $c26;"         // *
    "csplit $c26, $c2, $c26, $3;"  // *split the stack
    "cgetpcc $c1;"                 // get pcc (code ret cap for callee)
    "cincoffset $c1, $c1, 20;"     // properly set return address
    "cseal $c1, $c1, $c3;"         // *seal the code capability
    "cseal $c2, $c2, $c3;"         // *seal the data capability
    "daddi $6, $6, -1;"            // n-1
    "j fac;"                       // *fac(n-1)
    "nop;"                         // branch delay
    "csplice $c26, $c4, $c26;"     // *put the stack back together
    "cld $6, $zero, 0($c26);"      // load n from the stack
    "clc $c1, $zero, 32($c26);"    // load the code return cap from the stack
    "clc $c2, $zero, 288($c26);"   // load the data return cap from the stack
    "daddi $3, $zero, 772;"
    "cincoffset $c26, $c26, $3;"   // clean up the stack
    "mult $6, $2;"                 // multiply n and the result of the fac call
    "mflo $2;"                     // put the result as return value
    "cmove $c4, $c26"              // *return our stack ($c26 will be overwritten)
    "ccall $c1, $c2, 1;"           // *return
    "nop;"
);
```

Future work
- More extensive evaluation
- Compiler support
- Concurrency

VRIJE UNIVERSITEIT BRUSSEL
Software Languages Lab

Aaron Joos Lippeveldts
aaron.joos.lippeveldts@vub.be