Design and Implementation of a Dynamic Information Flow Tracking Architecture to Secure a RISC-V Core for IoT Applications

Christian Palmiero†, **Giuseppe Di Guglielmo•**, Luciano Lavagno†, Luca P. Carloni•

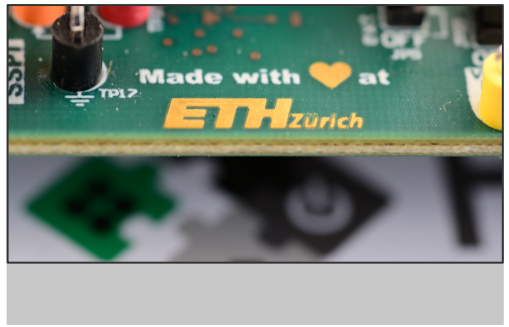
† *Politecnico Di Torino*

• *Columbia University*

2018 IEEE High Performance Extreme Computing Conference

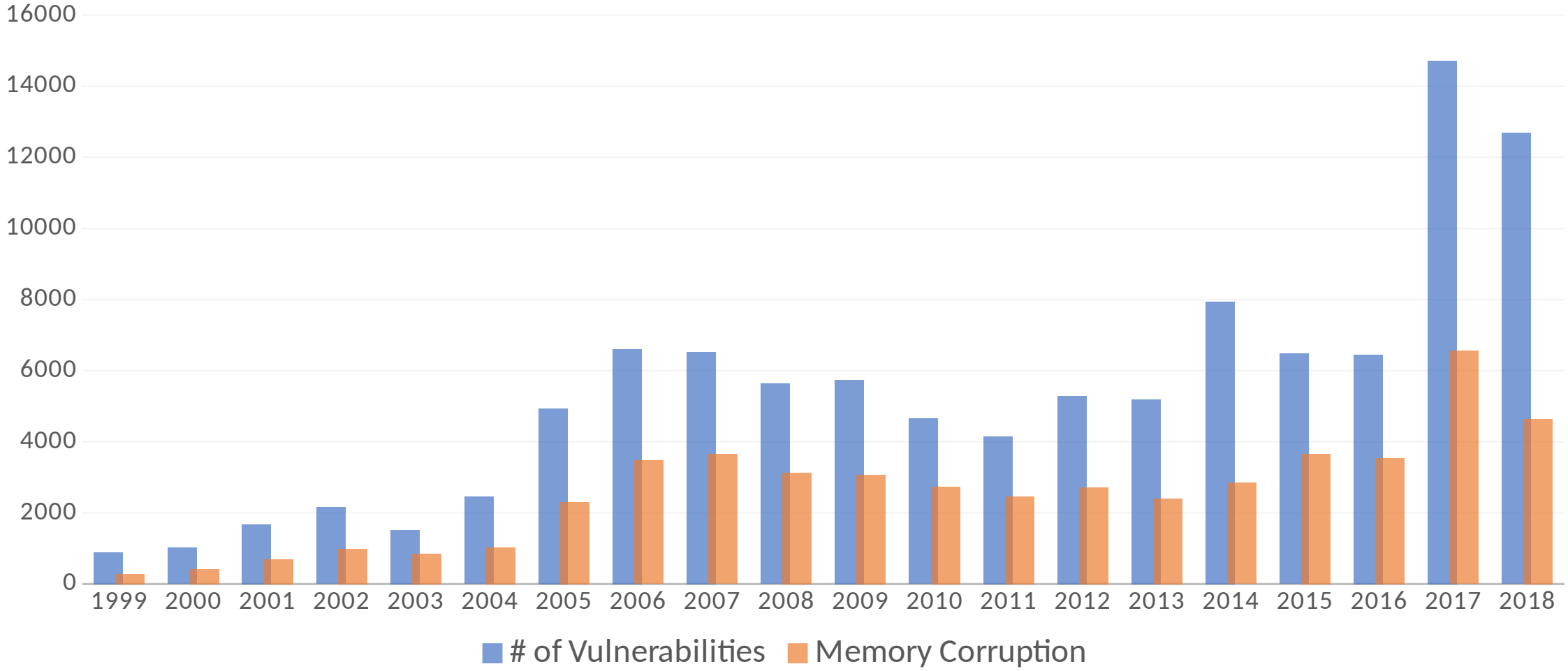
Trend #1: Open Source Hardware

- RISC-V is an open Instruction Set Architecture
  - It is not a company or a processor implementation
  - RISC-V Foundation (2015)
    - Non profit – To guide future development of the architecture
    - 100 members: Google, NVIDIA, Qualcomm, and Samsung …
    - RISC-V Workshop, RISC-V Meetup, RISC-V Day, RISC-V Summit
  - RISC-V creators formed a startup (SiFive) to design custom RISC-V chips for customers
    - Processors (embedded, OS-capable), IP, SoC, tools,…
    - Raised $64.1 Million
    - Western Digital had signed a multi-year license and had pledged to produce a billion RISC-V cores
    - Partner with NVIDIA for Deep Learning SoC
  - PULP project of ETH Zurich and University of Bologna
    - Focus on parallel, ultra-low-power, and embedded
    - 27 prototype chips from 180nm to 22nm



Trend #2: Importance of Software Security

- From the US National Vulnerability Database

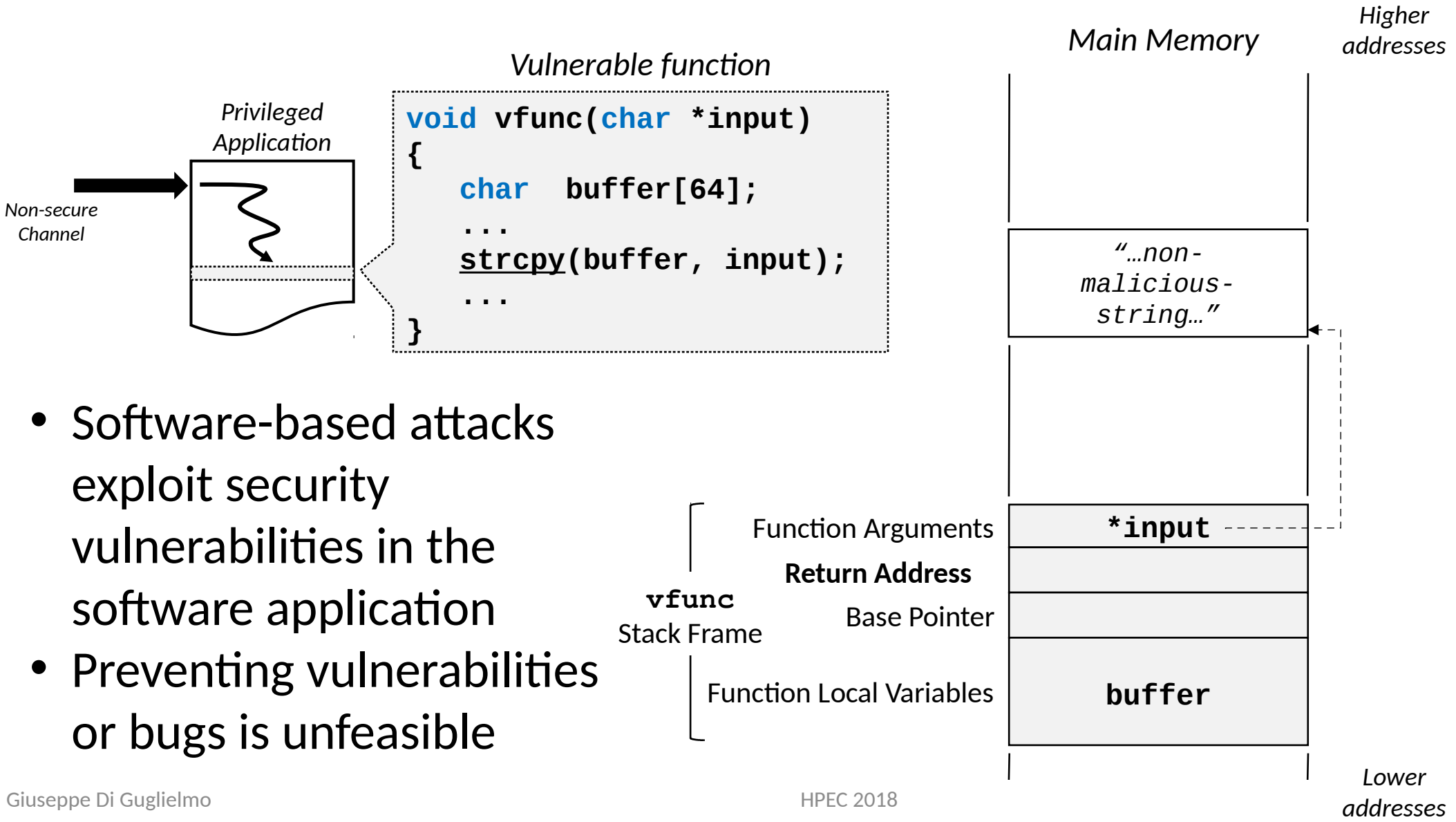


Research Question

- How can we protect software running on a RISC-V core against the most common software vulnerabilities?

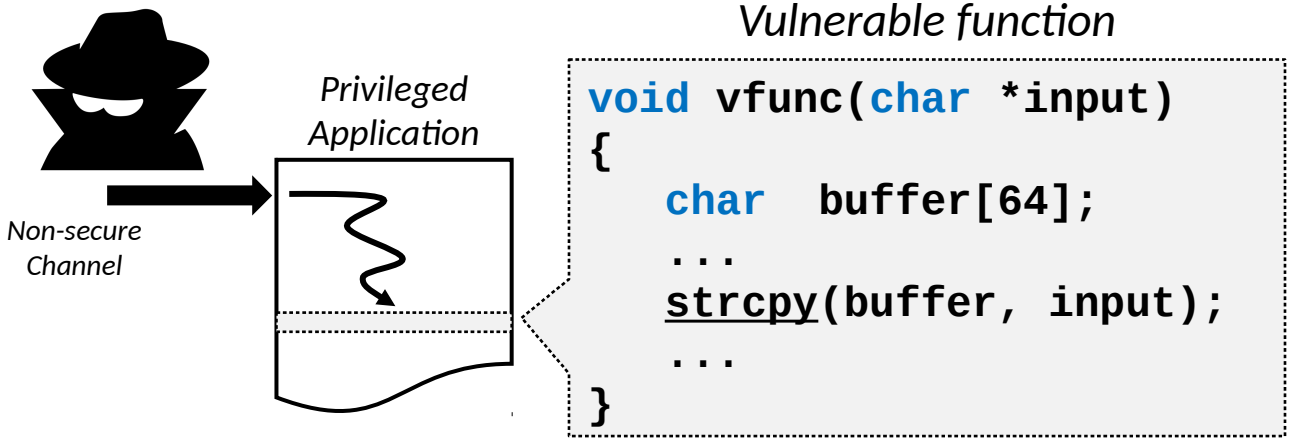
- The protection scheme has to be
  - Able to detect and stop memory-corruption attacks
  - Flexible and extendable
    - Software-programmable security policies to target future kinds of attacks
  - Transparent and fine-grain
    - No latency and reduced area overhead

A Vulnerable Application



- Software-based attacks exploit security vulnerabilities in the software application
- Preventing vulnerabilities or bugs is unfeasible

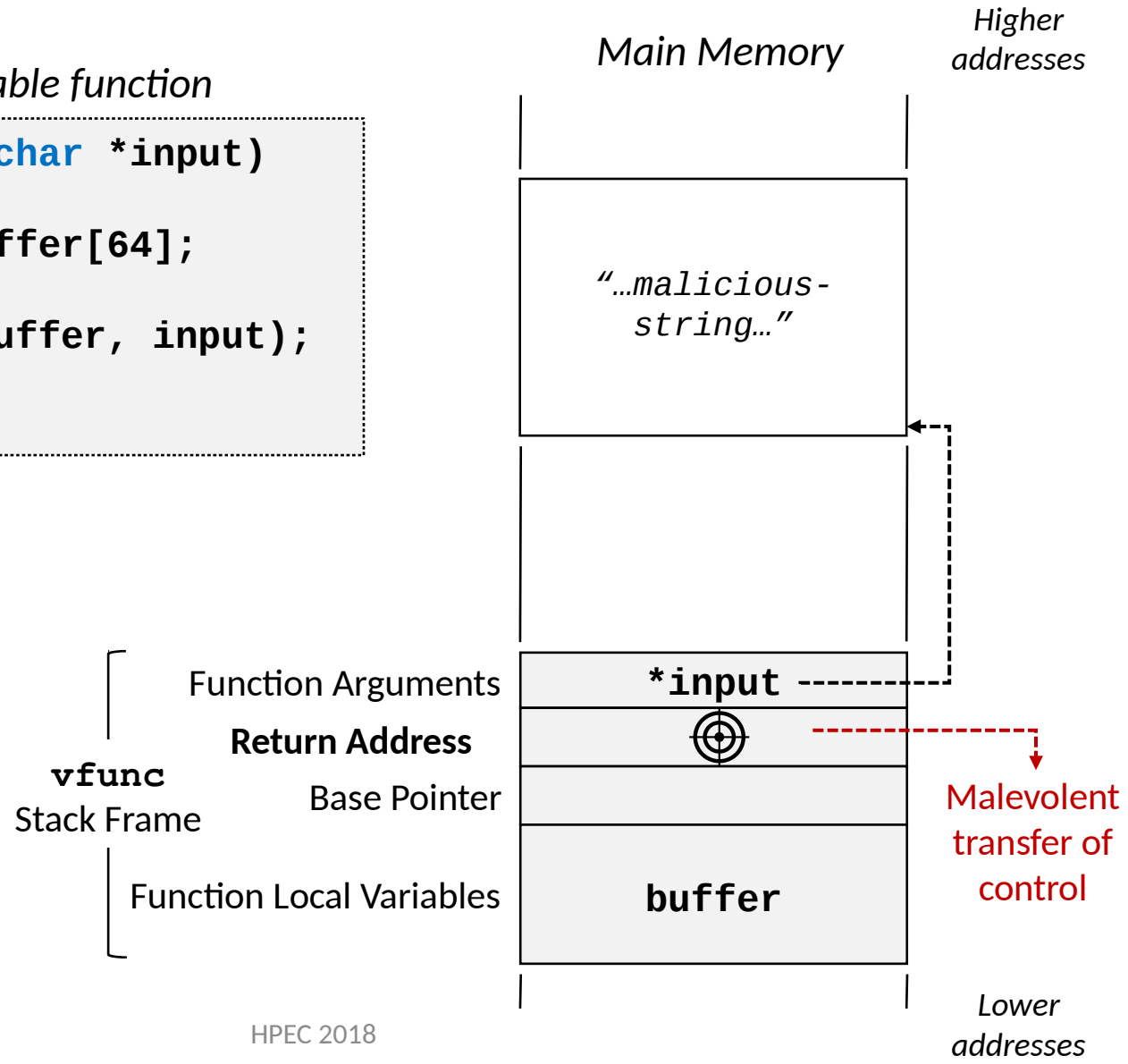
Buffer Overflow



Vulnerable function

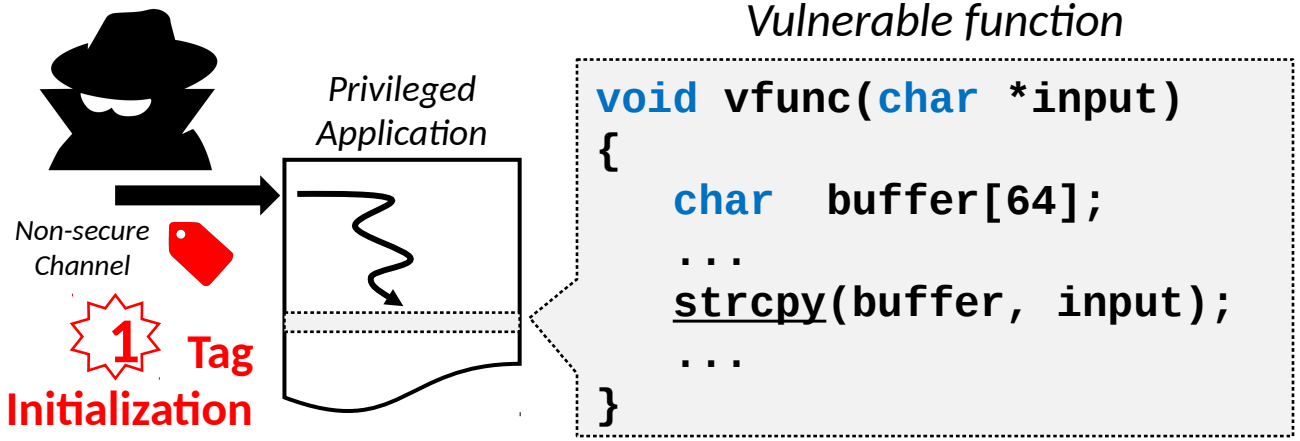
```
void vfunc(char *input)
{
   char  buffer[64];
   ...
   strcpy(buffer, input);
   ...
}
```

- Hijacking a privileged program is a security risk for the entire system
- Preventing vulnerabilities or bugs is unfeasible

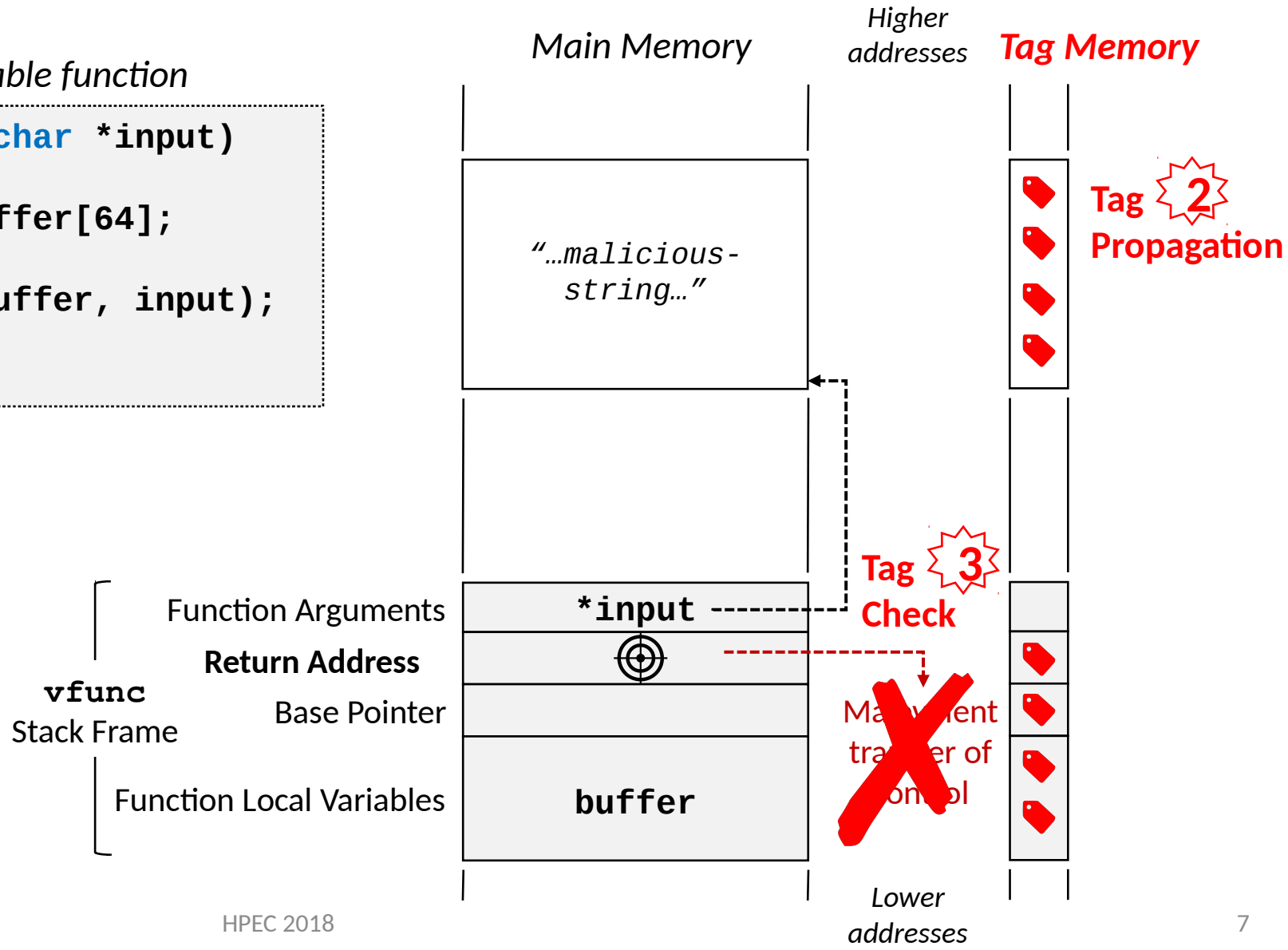


Dynamic Information Flow Tracking

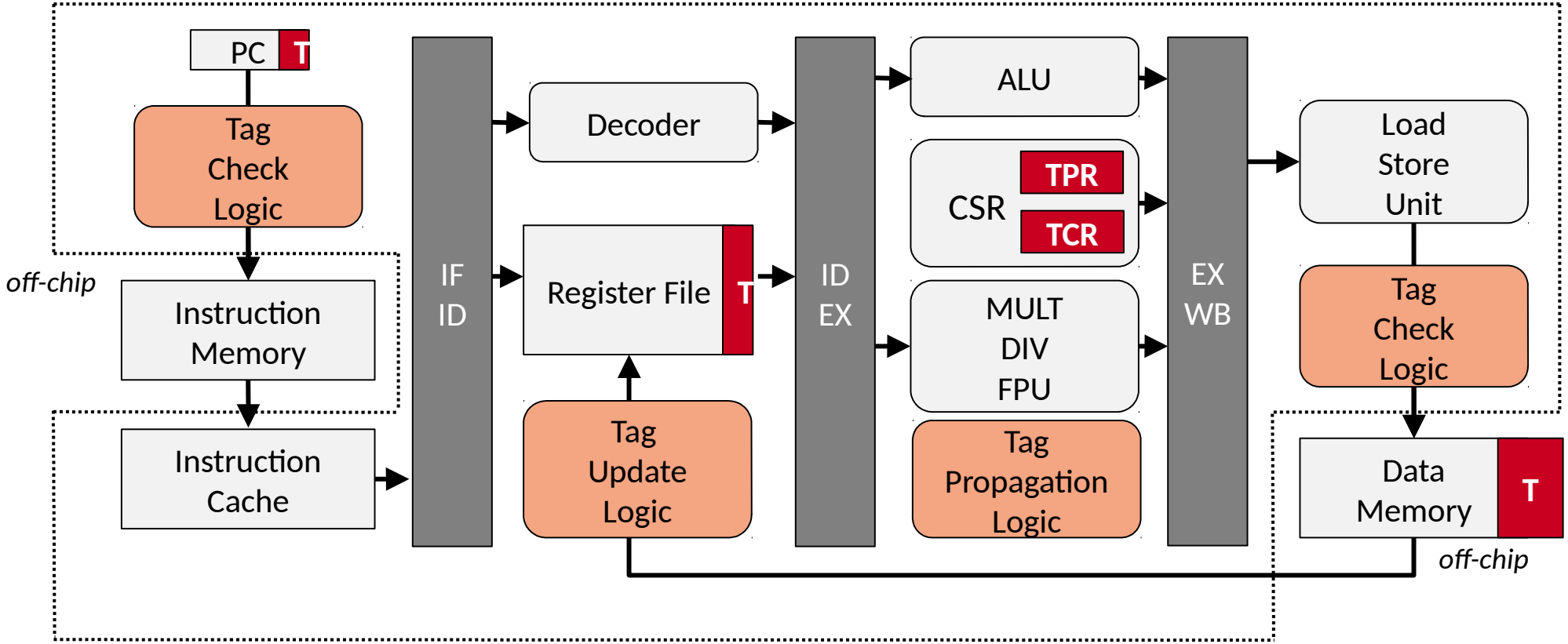
G. Edward Suh et al., *Secure Program Execution via Dynamic Flow Tracking*, 2004



- DIFT is a combination of mechanisms and policies to protect vulnerable programs against software attacks

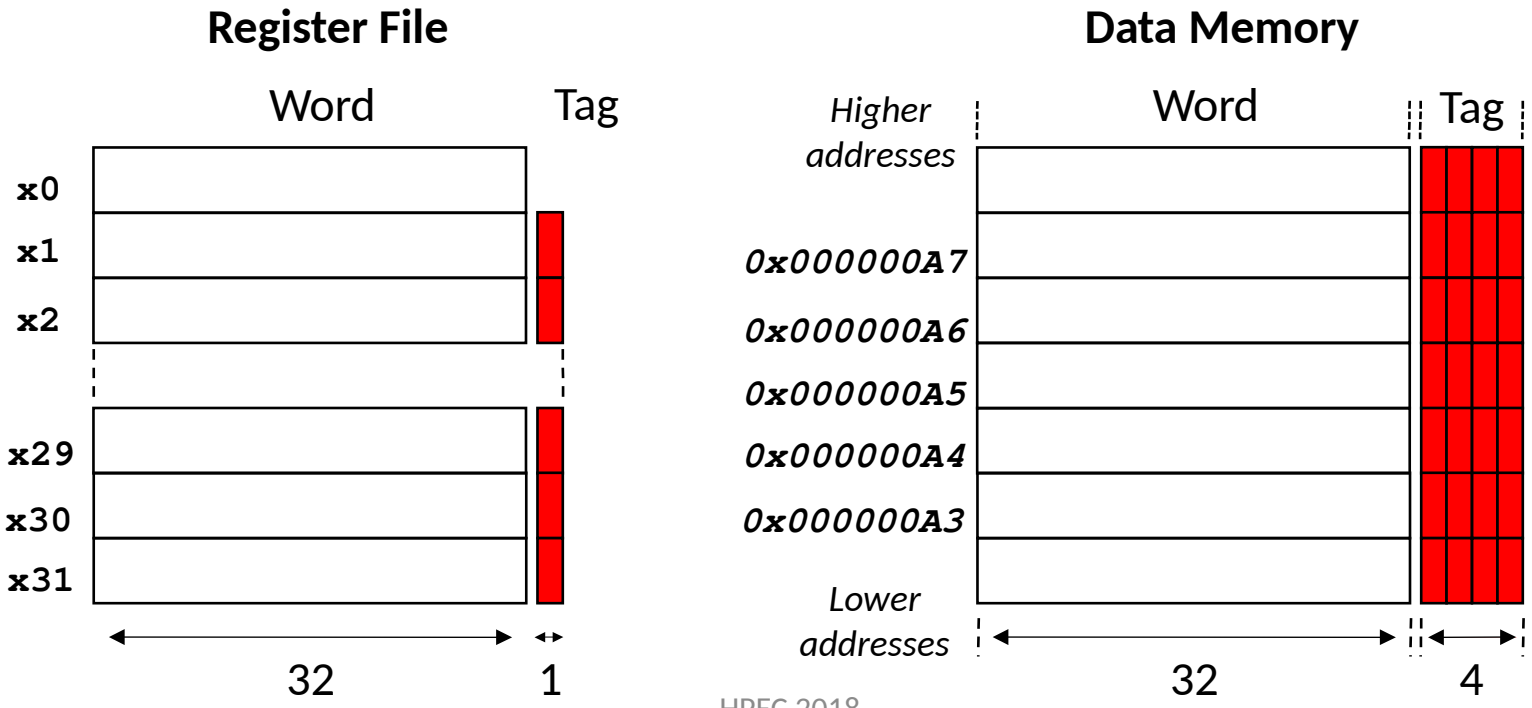


Securing RISC-V with DIFT



Tag-extended Memories (Mechanism)

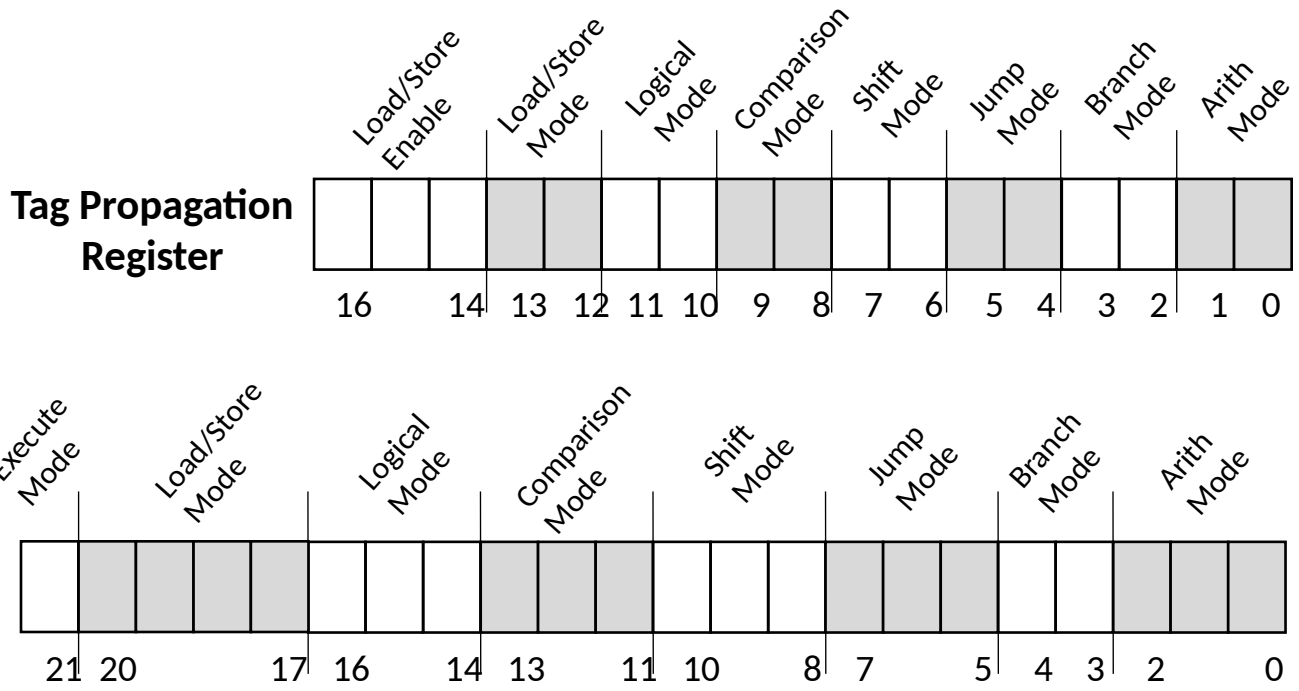
- Each data element is stored in memory with its tag
- To access both data and tag, we use the same index (memory address or register id)
- Coupled approach
  - The data and tag are always transmitted atomically
  - Extension of the data-memory bus from 32 bits to 36 bits



Tag-Propagation and Check (Mechanism)

- We organize the instruction in classes to increase the flexibility of the protection scheme
- We added tag-propagation and check registers (TPR, TCR) to the control status register (CSR)
  - TPR and TCR store the propagation and detection rules

CLASS	INSTRUCTIONS
Load/Store	LW, LH[U], LB[U], SW, SH, SB, LUI, AUIPC, XPulp Load/Store
Logical	AND, ANDI, OR, ORI, XOR, XORI
Comparison	SLTI[U], SLT[U]
Shift	SLL, SLLI, SRL, SRLI, SRA, SRAI
Jump	JAL, JALR
Branch	BEQ, BNE, BLT[U], BGE[U]
Integer Arithmetic	ADD, ADDI, SUB, MUL, MULH[U], MULHSU, DIV[U], REM[U]



Programming the DIFT-protected RISC-V (Mechanism)

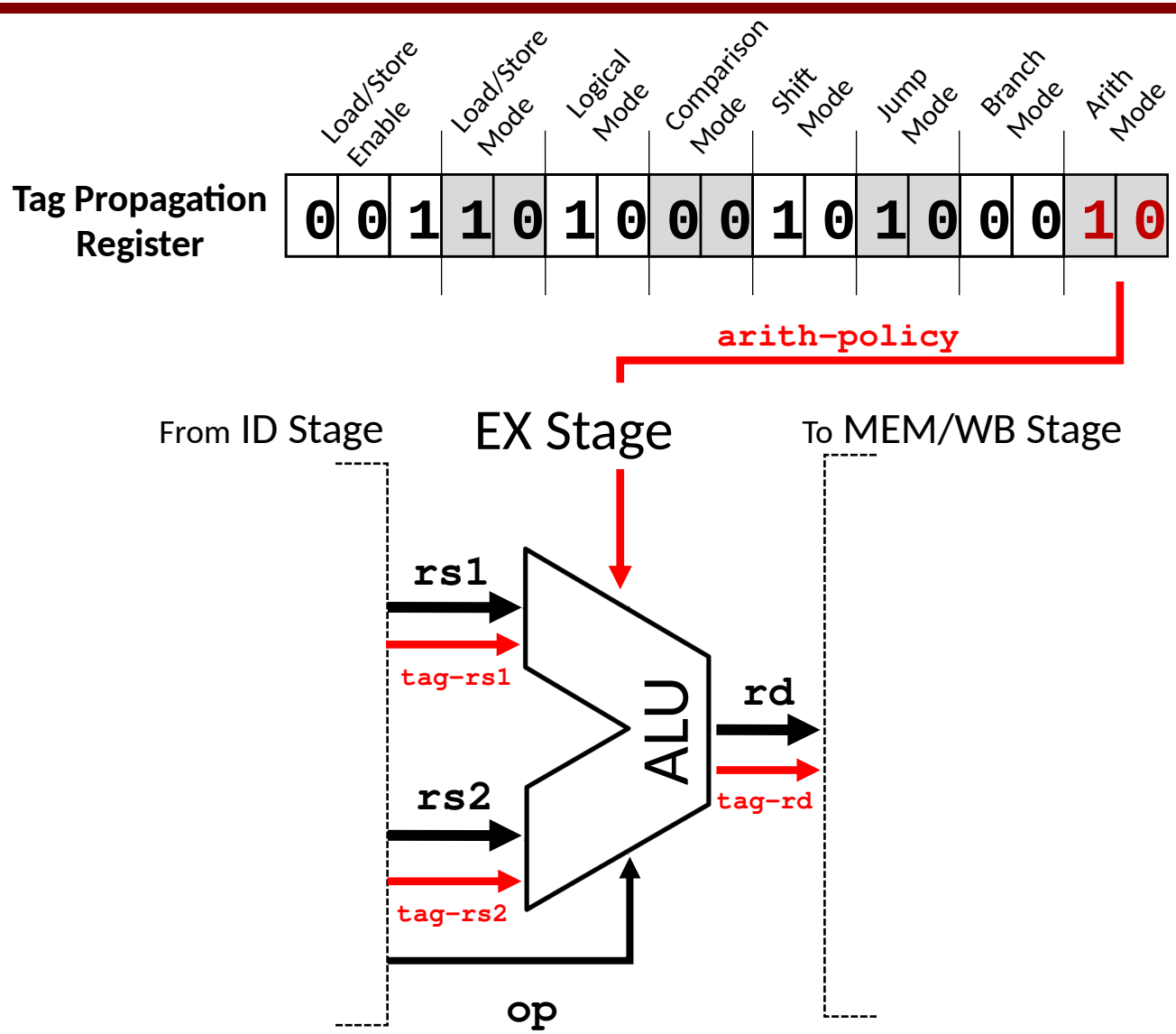
- Programmable hardware scheme
  - To tag non-secure channels as spurious we introduce new instructions
    - mark as spurious a register or a byte/half-word/word in memory
  - To configure TPR and TCR we use a startup routine before the **`main()`** function
- Because we run without OS protection, we assume that all of the I/O channels are untrusted
  - For example memory-mapped peripherals

```
#define SIZE 32
void tag_words(u32 *data_ptr, u32 size) {
  for (u32 i = 0; i < size; i++) {
    /* p.spsw set a tag for each byte in a
     * memory word */
    asm volatile ("p.spsw x0, 0(%[offset]);"
                  :
                  :[offset] "r" (data_ptr);
    data_ptr++;
  }
}

void vfunc(u32 input_1[SIZE],    /* non-secure */
           u32 input_2[SIZE],    /* non-secure */
           u32 input_3[SIZE]) { /* secure */

   /* Tag initialization phase*/
   tag_words(SIZE, input_1);
   tag_words(SIZE, input_2);

   /* Function body */
   /* ... */

}
```

Tag-Propagation Policies

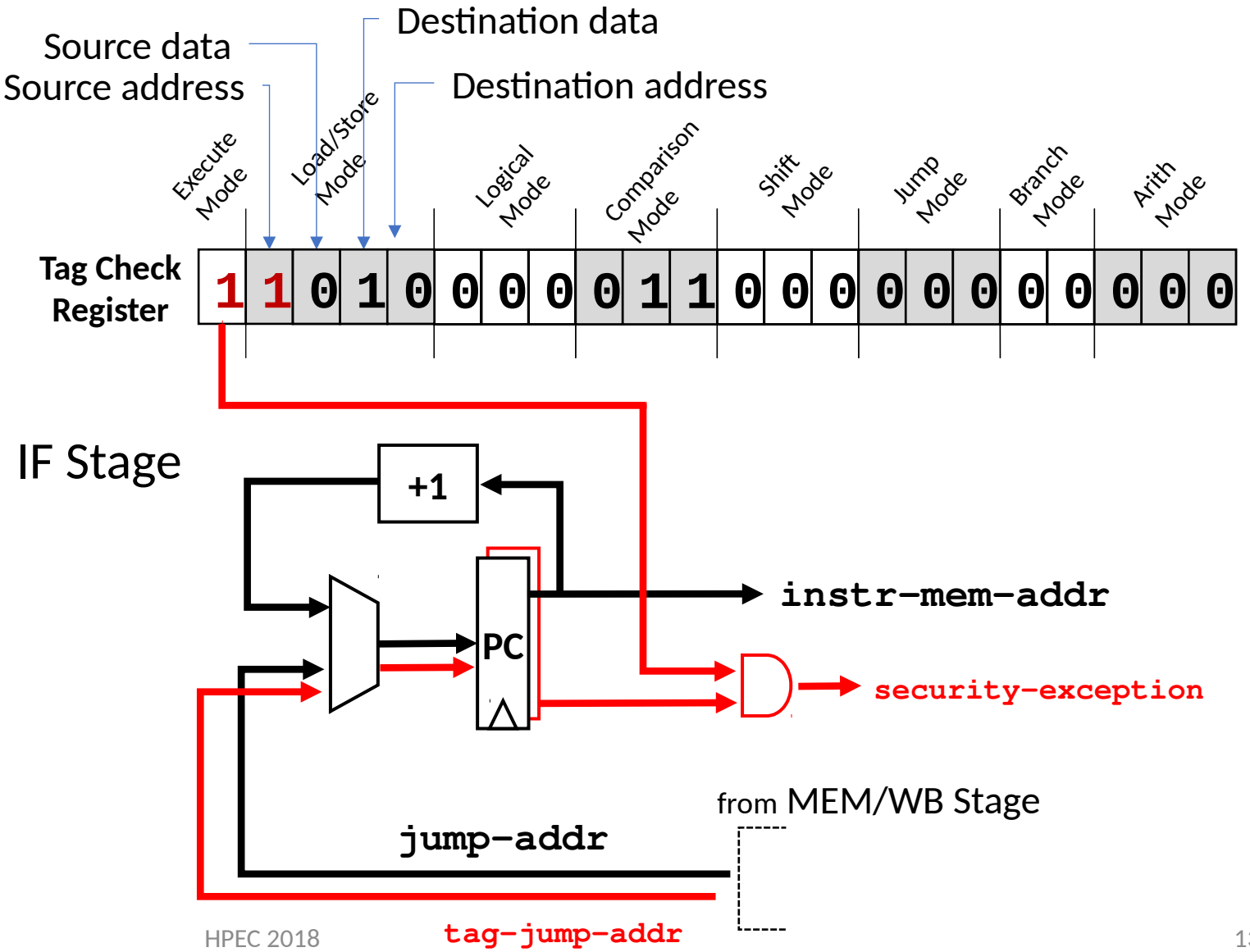
- Define how tag values must be propagated from input operands to output operand of an instruction
  - TPR modes
    - 00: keep the old tag value
    - 01: the output tag is 1 if both the input tags are set
    - 10: the output tag is 1 if at least one input tags is set
    - 11: discard the tag (set tag to zero)
  - An example:

    "For an arithmetic instruction, if at least one input operand is tagged then the output is tagged"



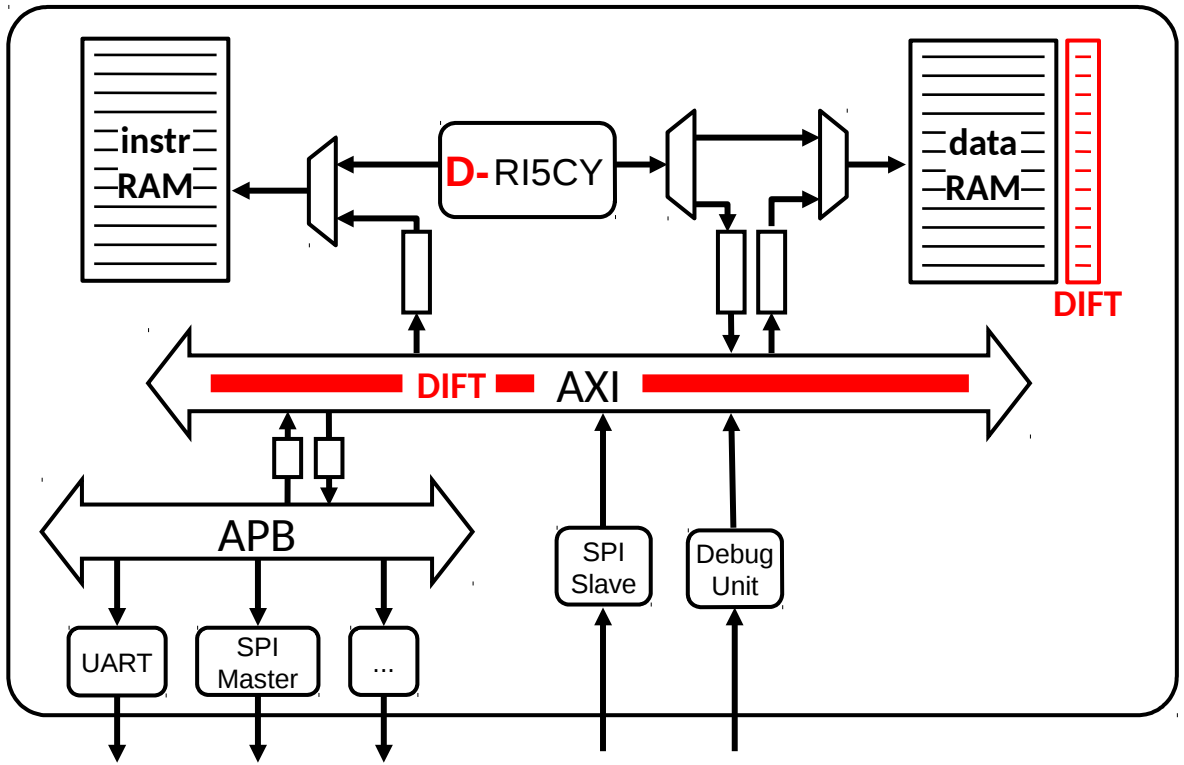
Tag-Check Policies

- Tag-check rules restrict the operations that may be performed on tagged data
- Some examples
  - "If the program counter is tagged, rise a security exception"
  - "If a register is tagged it cannot be used to address the data memory"



Experimental Setup

- We extended the RI5CY/PULPino implementation
  - Target FPGA
    - ZedBoard (Xilinx XC7Z020)
  - The overall data memory was extended from 32KB to 36KB (12.5%)
  - DIFT propagation on the interconnect uses the USER channels of the AXI4 standard
  - The overall increase in logic
    - 6% of the LUT w.r.t. RI5CY
    - < 1% of the LUT w.r.t. SoC



Methodology Validation

- J. Wilander and M. Kamkar's suite of buffer-overflow attacks (2003)
  - C language
  - Attacks were ported from x86 to RISC-V architecture

ATTACK #	LOCATION	TARGET	TECHNIQUE	RESULT
1	Stack	Return Address	Direct	Detected
2	Stack	Base Pointer	Direct	No False Positive
3	Stack	Function Pointer (local variable)	Direct	Detected
4	Stack	Function Pointer (function parameter)	Direct	Detected
5	Heap/BSS/Data	Function pointer	Direct	Detected
6	Stack	Return Address	Indirect	Detected
7	Stack	Base Pointer	Indirect	No False Positive
8	Stack	Function Pointer (variable)	Indirect	Detected
9	Stack	Function Pointer (function parameter)	Indirect	Detected
10	Heap/BSS/Data	Return Address	Indirect	Detected
11	Heap/BSS/Data	Base Pointer	Indirect	No False Positive
12	Heap/BSS/Data	Function Pointer (variable)	Indirect	Detected
13	Heap/BSS/Data	Function Pointer (function parameter)	Indirect	Detected

- TESO Hacker group – Paper on format-string attacks (2001)

ATTACK #	SOFTWARE	RESULT
1	GHTTPD 1.3/1.4	Detected
2	wu-ftpd 2.6.0	Detected

False-Positives Analysis

- We chose the PULPino regression suite
  - 2d Convolution, AES, Discrete Cosine Transform, Fast Fourier Transform, Finite Impulse Response, Inflection Point Method, Matrix Multiplication, Keccak/SHA-3
- For example, we marked as spurious the two input matrices of Matrix Multiplication
  - The result will be spurious as well
  - But it does not rise security exception because those values are never used in an unsafe manner
    - E.g. as program counter value or load/store source/destination addresses

Conclusions

- D-RI5CY: DIFT-secure RISC-V core
  - Software programmable policy
  - Fast and transparent
    - No run-time overhead
    - 1% area overhead, 12.5% data-memory overhead
  - Easily extended to target new set of attacks
  - Validated on security suites that we adopted and extended from the literature

- This work is part of a broader research activity on securing Heterogeneous SoC
  - PAGURUS: Low-Overhead Dynamic Information Flow Tracking on Loosely Coupled Accelerators
    - IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
    - Will be presented at CODES+ISSS 2018

From the Press Room

September 17, 2018
(one week ago…)

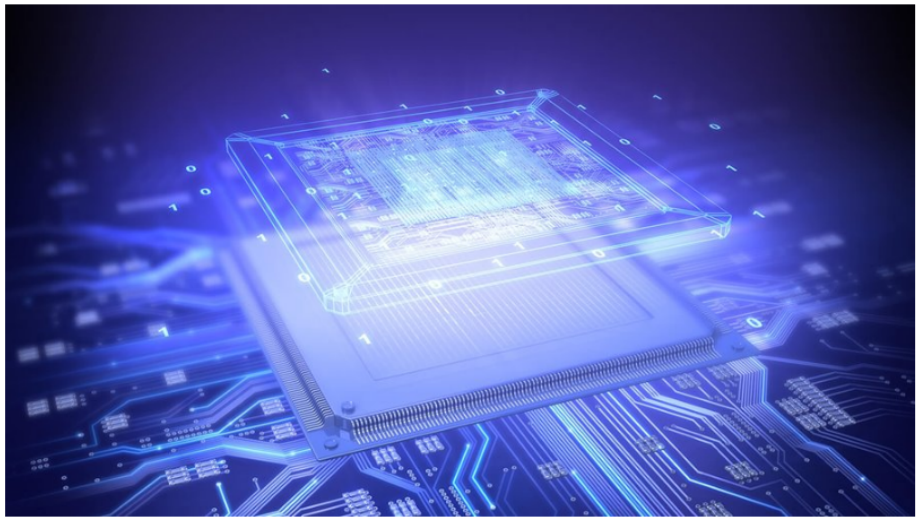
arm Community

Search

Communities Blog Forums Activity Support

Processors › Processors blog

Arm A-Profile Architecture Developments 2018: Armv8.5-A



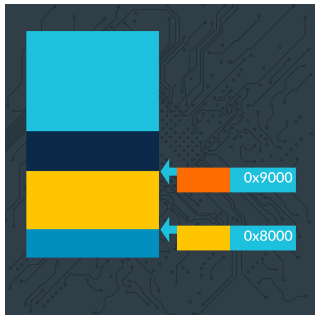
Security: Vulnerability Detection

The discovery of Spectre and Meltdown has dominated the security narrative for the past 12 months. However, these are not the only security challenges we face today, and Arm has been working with its partners on developing hardware tools to improve software resilience to attacks.

Many of the most common software vulnerabilities are caused by buffer overruns, and use-after-free coding errors. Buffer overruns refer to an instance where a program tries to access memory beyond its intended range. Use-after-free errors occur when a program accesses memory after it has freed it for reuse. Often, these errors are situation-dependent, requiring specific circumstances in order to occur. Famously, the Morris worm in 1988 was the first documented use of a buffer overrun for malicious purposes. Thirty years later, we are still facing the same software issues.

Memory Tagging

Armv8.5-A incorporates a new feature called Memory Tagging. When Memory Tagging is in use, a tag is assigned to each memory allocation. All accesses to memory must be made via a pointer with the correct tag. Use of an incorrect tag is noted and the operating system can choose to report it to the user immediately, or to note the process in which it occurred, for later investigation.



Security: Limiting Exploits

Once an attacker has found a vulnerability to exploit, their next aim is to execute code to gain control of the machine they have accessed. Techniques used include ROP and JOP Attacks (Return- and Jump-Oriented Programming). These techniques find small sections (called gadgets) of vulnerable programs that chain together to run the code the attacker wants. These methods work because the architecture puts no restrictions on where code can branch to, or where branches can have come from. This enables attackers to use small snippets of functions, which do what they want.