Security of embedded Systems

Peter Langendörfer

telefon: 0335 5625 350
fax: 0335 5625 671

e-mail: langendoerfer [at] ihp-microelectronics.com
web: http://www.tu-cottbus.de/fakultaet1/de/sicherheit-in-pervasiven-systemen/

Organizational stuff

- Exam: most probably oral
- Lecture schedule subject of change due to „Bahn issues“
- No lectures at:
  - October 22
  - November 5

- Exercises on demand: Dr. Z. Dyka
  - dyka@ihp-microelectronics.com

Outline

- Introduction
- Design Principles
- Memory Management recap
- Attacks
- Protection means
  - Hypervisor/Micro kernels
  - Canaries
  - Isolation
  - Access control/rights management
- Code Attestation
- Secure Code Update

Control flow attacks

- Can be used for:
  - Bypassing security controls
  - Mal-Packets (self-propagating malicious packets)
  - Code injection
- Code injection is a very powerful attack allowing:
  - A worm to propagate
  - An attacker to take complete control of a device
  - Installing a rootkit
  - Changing / disclosing cryptographic keys

Standard Code Injection Techniques

- Overflow a buffer on stack (or other memory corruption…)
  - Store instructions in the buffer
  - Overwrite the return address
  - Move the program counter to those instructions (in the stack)

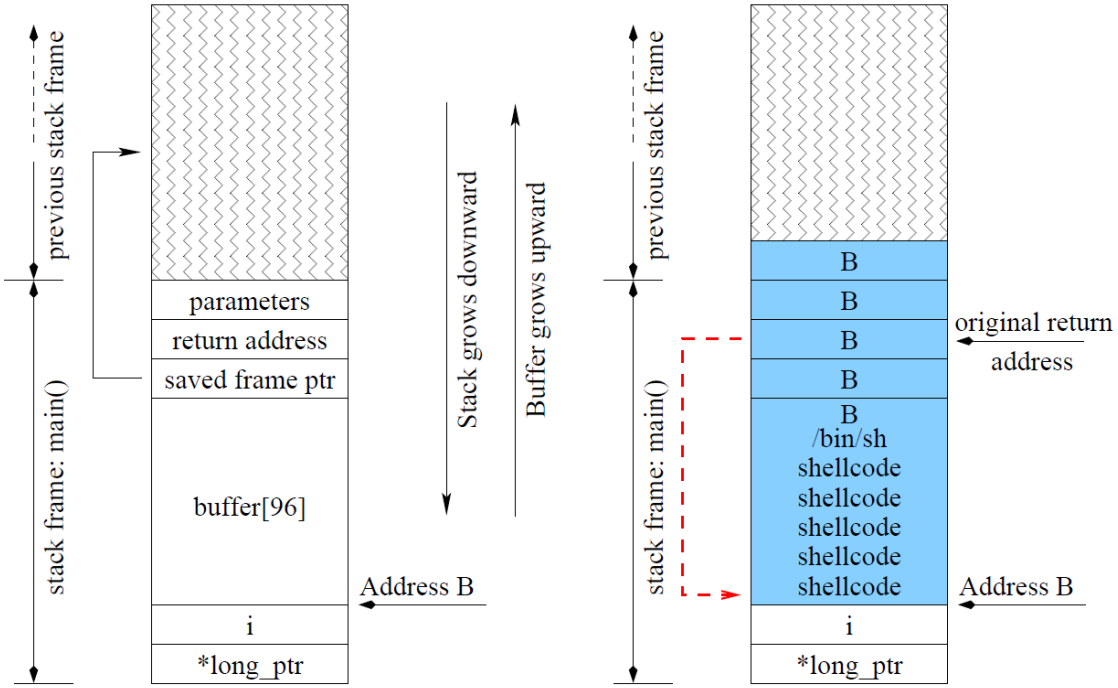
- Possible on Von Neumann architecture
  - Can be prevented by making Stack Memory non executable(W xor X)
- Not possible on Harvard Architectures

- Return to libc attacks to « bypass » W xor X
- Return Oriented Programming : a more general approach

Recap Buffer Overflow

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
  "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
  "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
char large_string[128];
void main() {
  char buffer[96];
  int i;
  long *long_ptr;

  long_ptr=(long *)large_string;
  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
  for (i=0; i<strlen(shellcode); i++)
    large_string[i] = shellcode[i];
  strcpy(buffer,large_string);
}
```



Preventing Buffer Overflows

- Use safe programming languages, e.g., Java
  - Legacy C code? Native-code library implementations?
- Black-box testing with long strings
- Mark stack as non-executable
- Randomize memory layout or encrypt return address on stack by XORing with random string
  - Attacker won't know what address to use in his string
- Run-time checking of array and buffer bounds
  - StackGuard, libsafe, many other tools
- Static analysis of source code to find overflows

Example: Stack Guard

- StackGuard places a “canary” word next to (prior) the return address on the stack.
- Once the function is done, the protection instrument checks to make sure that the canary word is unmodified before jumping to the return address.
- If the integrity of canary word is compromised, the program will terminate.

Example: StackGuard

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.
  - Minimal performance effects: 8% for Apache.
- Newer version: PointGuard.
  - Protects function pointers and setjmp buffers by placing canaries next to them.
  - More noticeable performance effects.

- Note: Canaries don’t offer fullproof protection.
  - Some stack smashing attacks can leave canaries untouched.

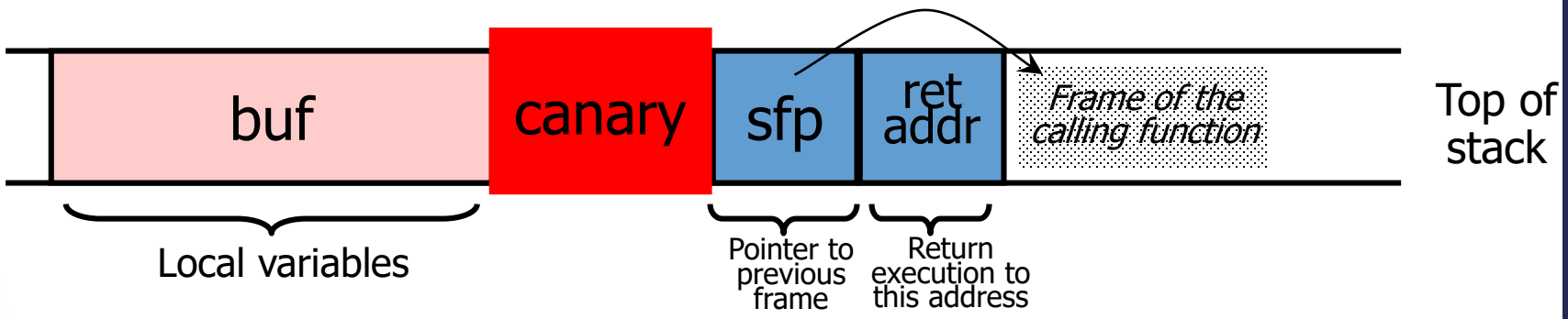
Canary Types

- Random canary:
  - Choose random string at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
  - To corrupt random canary, attacker must learn current random string.

- Terminator canary:
  - Canary = \0, newline, linefeed, EOF
  - String functions will not copy beyond terminator.
  - Hence, attacker cannot use string functions to corrupt stack.

StackGuard: Run-Time Checking

- Embed "canaries" in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

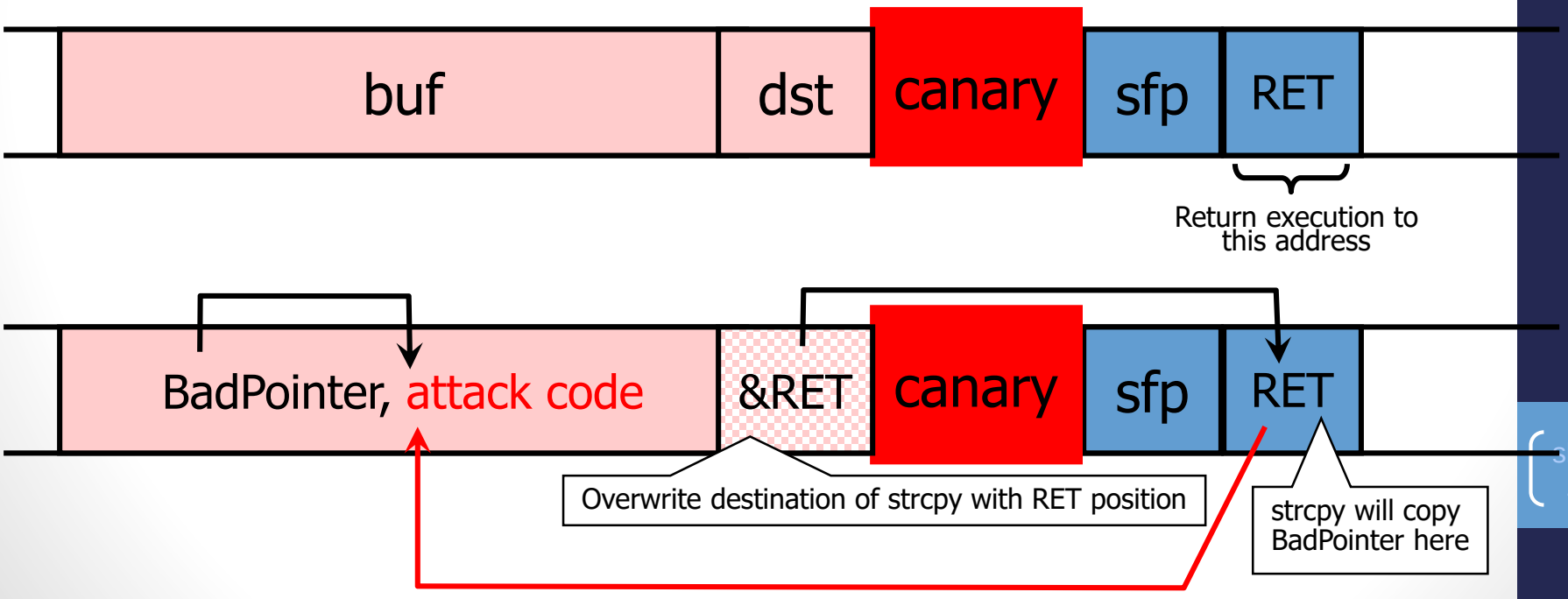


StackGuard Implementation

- StackGuard can be defeated
  - A single memory copy where the attacker controls both the source and the destination is sufficient
  - Vulnerability report
    - “BYPASSING STACKGUARD AND STACKSHIELD”, Phrack 56
    - “Four different tricks to bypass StackShield and StackGuard protection”

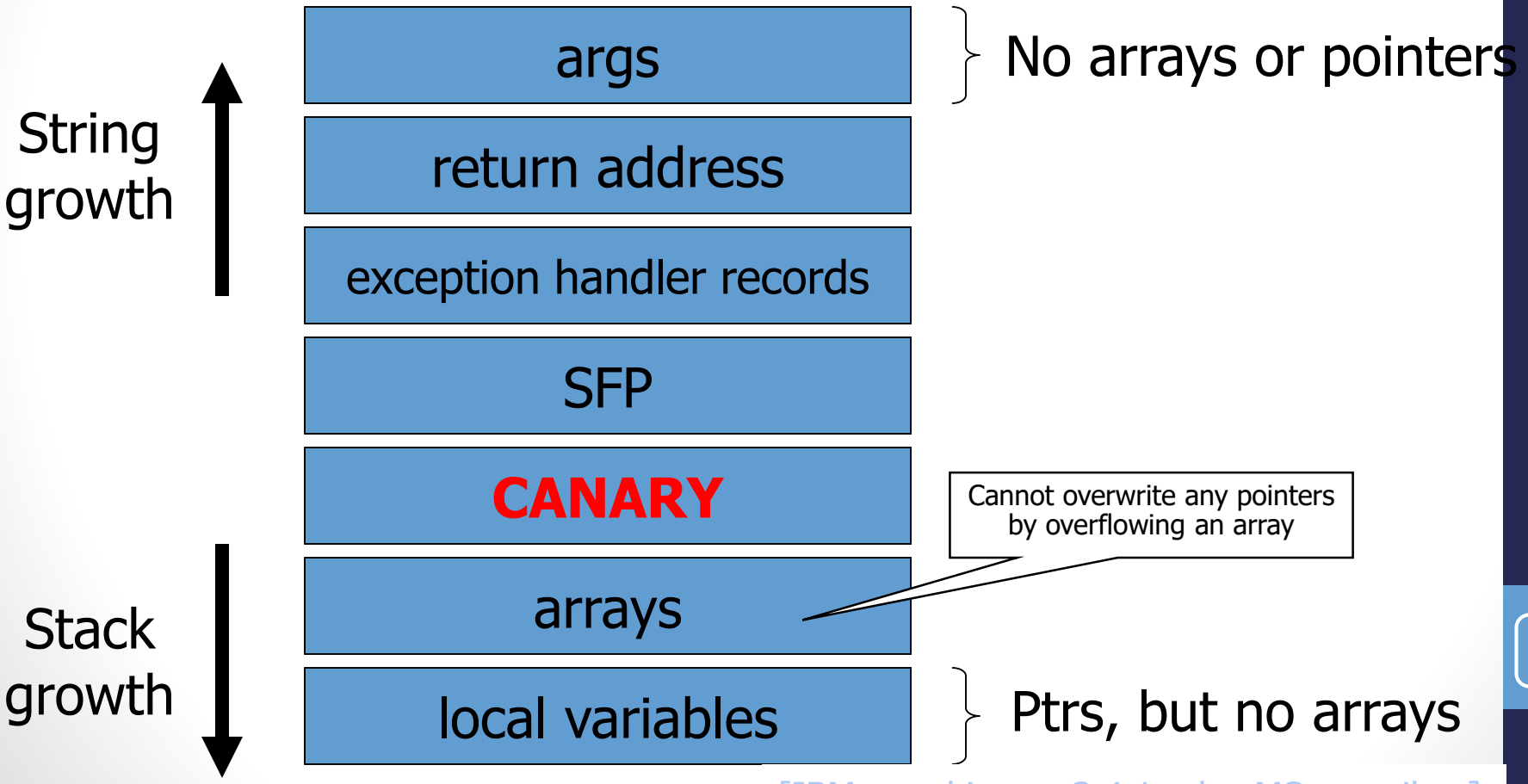
Defeating StackGuard

- Suppose program contains strcpy(dst,buf) where attacker controls both dst and buf
  - Example: dst is a local pointer variable



Example ProPolice / SSP

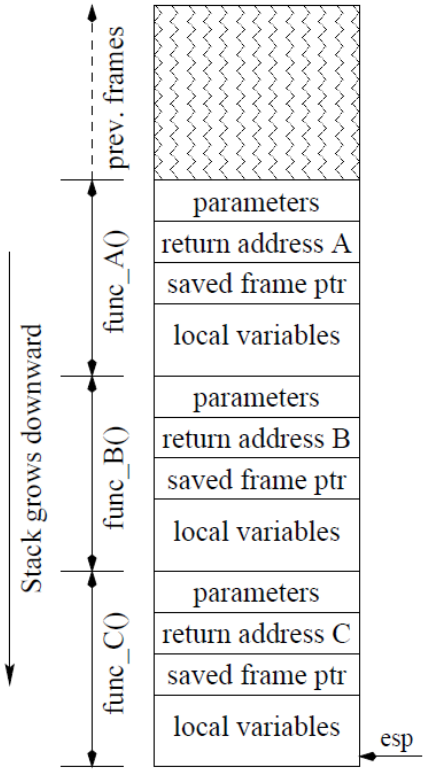
- Extends the canary idea of StackGuard
- Rearrange stack layout (requires compiler modification)

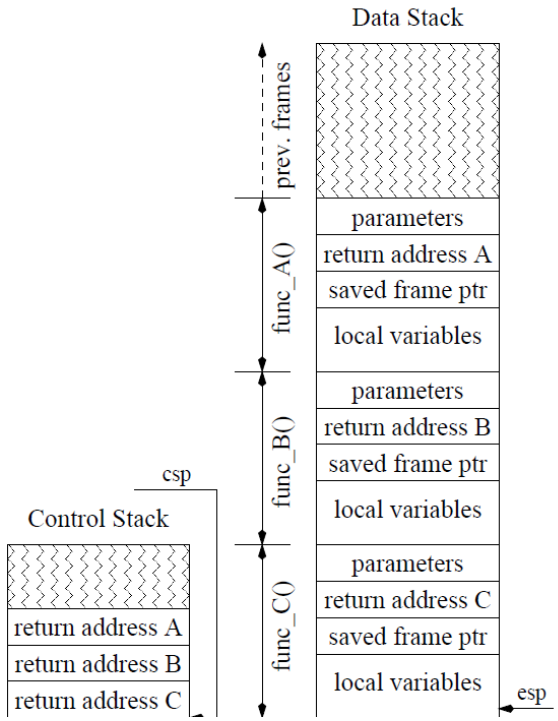


[IBM, used in gcc 3.4.1; also MS compilers]

Split stack approaches

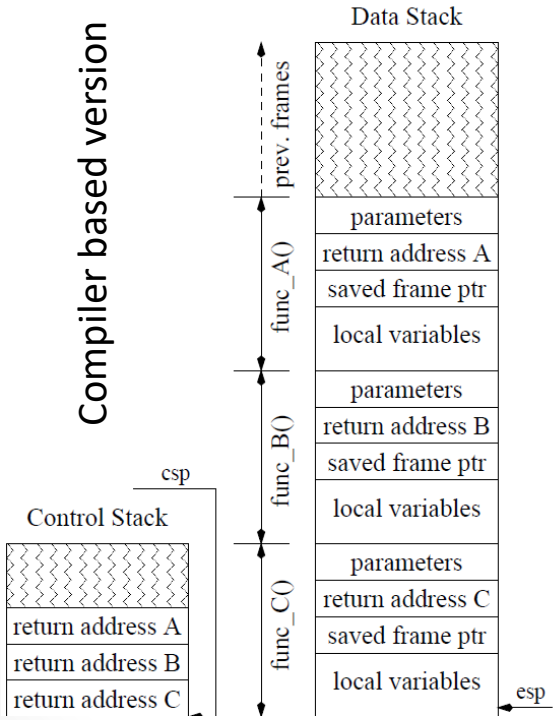
- Complier based approach:
  - prolog and epilog of each function call are modified by the compiler
  - Return addresses are written and read from the control stack
  - Allocation of the control stack “ensures” its integrity
  - Performance issues due to additional operations

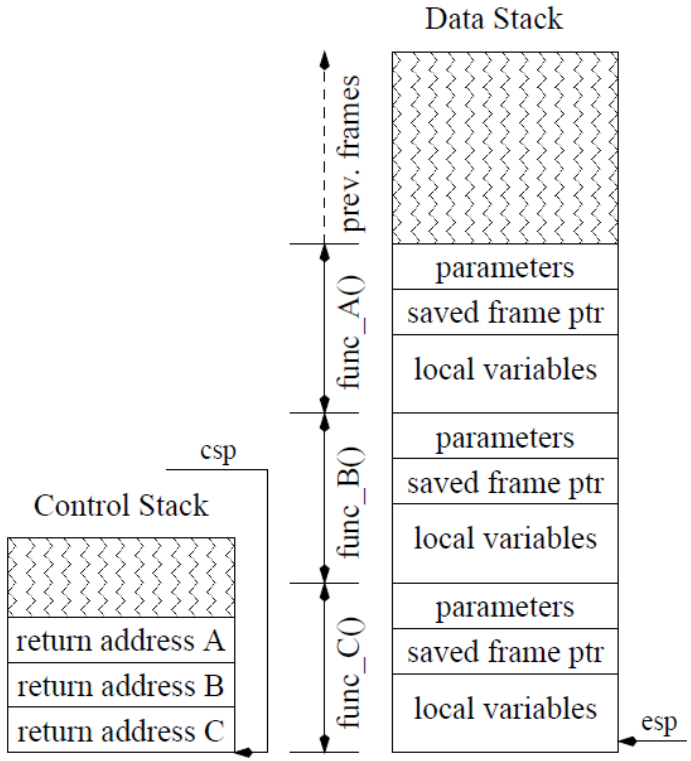




Split stack approaches

- Architecture based approach
  - 1 additional register „control stack pointer register (csp)“
  - Operations modified:
    - Call: to Push the function return address onto the control stack pointed to by csp,
    - Ret: to pop off the current top of the control stack (pointed to by the control stack pointer csp)





Split stack approach for embedded systems

- Ideal to prevent control flow manipulation:
  - Split the stack : control flow stack / data stack
  - Prevent manipulation of the control flow stack
- The data stack is manipulated as usual
  - Stack pointer can be moved (memory allocation)
  - Data can be read/write to the data stack
- Access to the control flow stack is restricted
  - Only call and ret instructions can be used to
    - Read/write on the CF stack
    - Modify the CF stack pointer

Split stack approach for embedded systems

- Some software solutions are possible but
  - Require lots of resources
  - Complete rebuild
- Here hardware control flow enforcement technique
- Implemented/tested both with
  - A modified VHDL implementation on a FPGA
  - Software on a modified simulator (AVRORA)

Control flow modification operations

- In the Atmel AVR core the program counter (PC) is
  - not accessible as a general purpose register
  - instructions such as load and store cannot modify it.

- On the AVR the following instructions can modify the control flow:
  - *Branch* and *Jump* (JMP) instructions update the control flow. However, as the destination address is provided as an immediate constant value, they are not vulnerable to manipulation and no return address is stored on the stack.
  - *Call* and *Return* instructions use the control flow stack pointer to access the control flow stack. Those instructions will store or fetch the control flow instructions on the control flow stack.
  - *Load* and *Store* instructions are prevented to alter the return stack, only access to data stack or other regions is allowed. The control flow stack and the data stack are checked to be non overlapping when a store is performed.

Control flow modification operations

- *Calli* instruction takes a function pointer as parameter (from a register). If the attacker is able to modify the pointer (or register) before it is used by an indirect call instruction, he would be able to control one control flow change but not the following ones.

- *Interrupts* transfer the control flow to a fixed interrupt handler and the address of the instruction that was executed while the interruption occurred is saved on the control flow stack, in this modified architecture the return address is therefore protected as well.

Memory layout stack memory areas configuration

- "Split Stack Configuration Register" (*SSCR*). In order to prevent the attacker from maliciously change this register configuration, it is made "writable once per boot": this configuration register is locked in hardware after the first write.

- addresses of the following configuration registers where chosen in the unused I/O registers addresses.

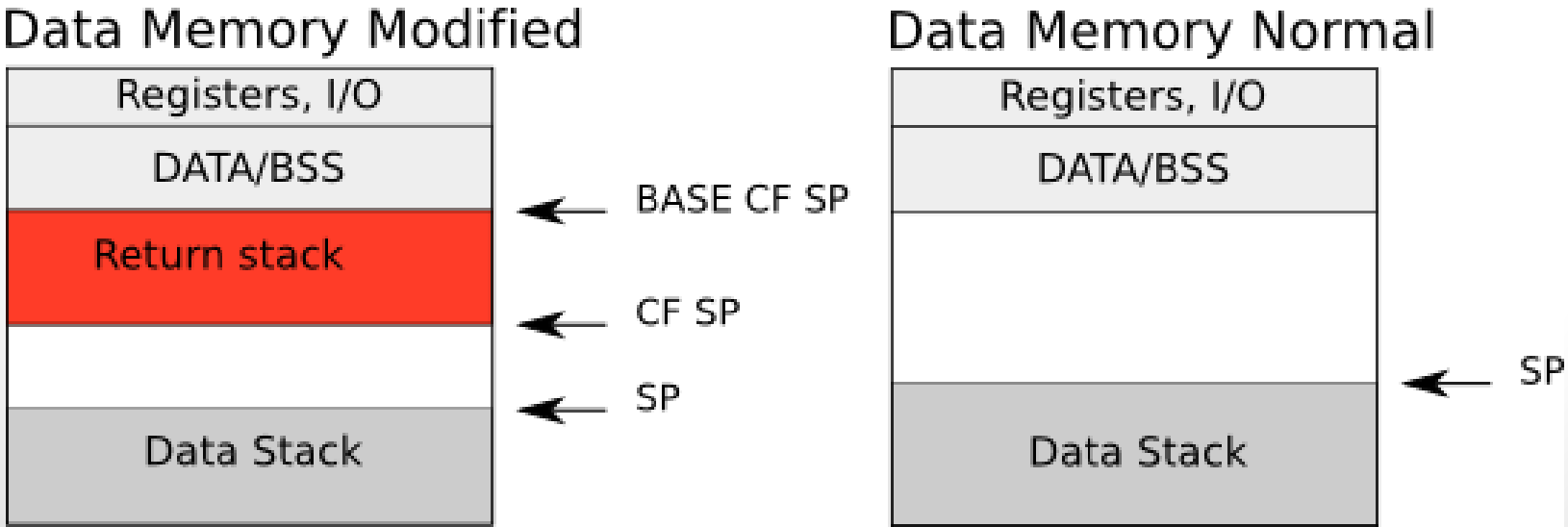
- memory layout configuration must be performed in order to enable the control flow stack and the memory access enforcement; new configuration registers:
  - *SSTACKEN* (Split STACK ENable) is a configuration bit which, when set, enables the split stack feature. It is part of the *SSCR* register.
  - *CF START* (Control Flow stack Start) is a configuration register used to fix the start of the control flow stack. It is automatically initialized from the libc to the end of the statically allocated memory (data/bss) therefore requires no user configuration.

Memory layout stack memory areas configuration

- new configuration registers:
  - *CF SP* (Control Flow Stack Pointer) is the control flow stack pointer. It is initialized with the same value as
  - *CF START* at boot and cannot be directly modified after initialization.
  - *CF STACK configured* is an internal signal in our modified core. It is automatically set after control flow registers have been set up. It cannot be modified by software and is reset when a reboot occurs. When this value is set any direct update of the *CF START* and *CF SP* registers are detected as possibly malicious modifications and therefore triggers a reboot.

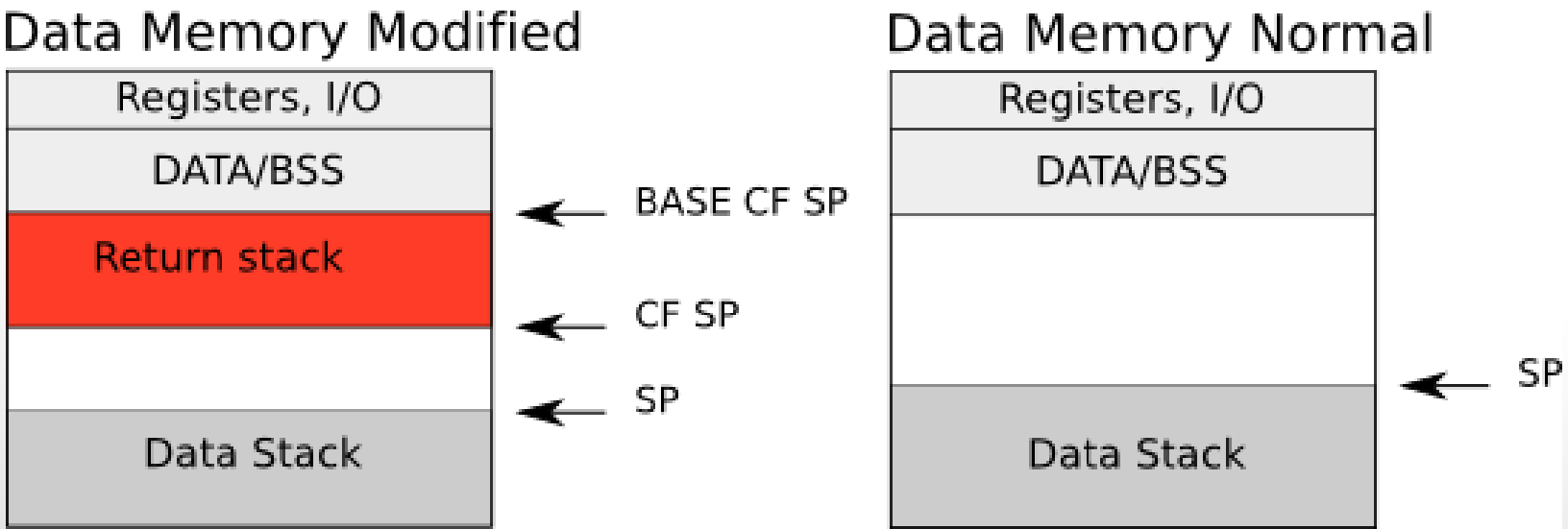
AVR modification

- Implemented in an AVR
  - VHDL code from opencores.org
- Modifications of the internal logic of call/ret
  - 2 stack pointers
  - Data stack start from end of memory
  - Control Flow stack starts from BASE_CF_SP



AVR modification

- Instruction Based Memory Access Control
  - Load/store can read write anywhere but the CF stack
  - Call / Ret access only CF stack
  - Base_CF_SP and CF_SP are locked after initialization
  - Stack pointers are checked at runtime when memory is accessed



AVR modification (II)

- Prevents :
  - Return address modifications (ret to libc …)
  - Return Oriented Programming even when malicious code is already present on the device (e.g. a “rootkit”)
  - Stack overflow (out of memory condition)

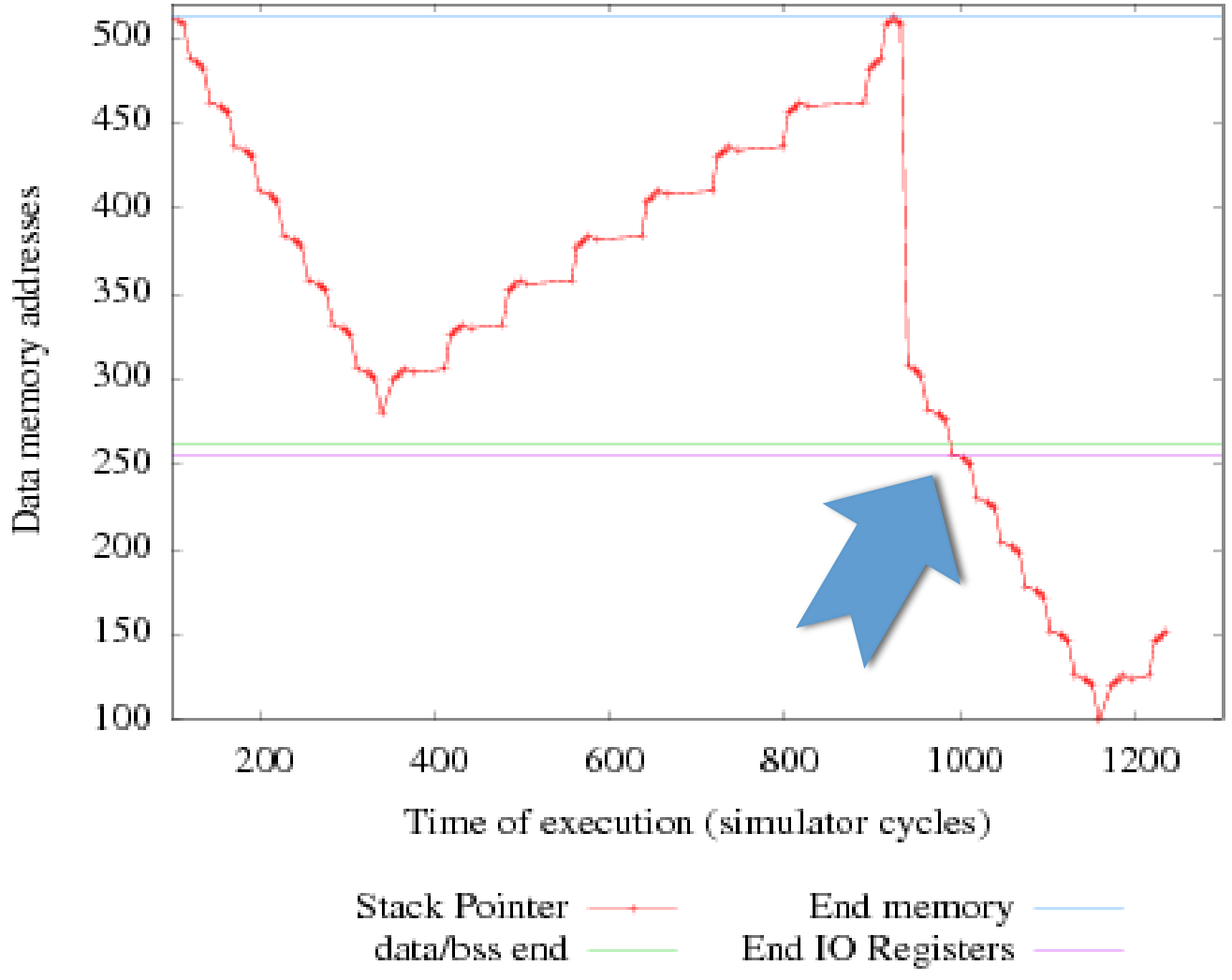
- Only software change :
  - initialization of CF_stack, in a library,
  - binary compatibility
  - Very low cost in terms of gates/complexity
  - No extra memory required
- Drawbacks
  - Can’t do context switching
  - Longjmp/setjmp
  - They would require a supervisor mode to be safe …
  - No prevention of non control attacks

Example stack overflow program

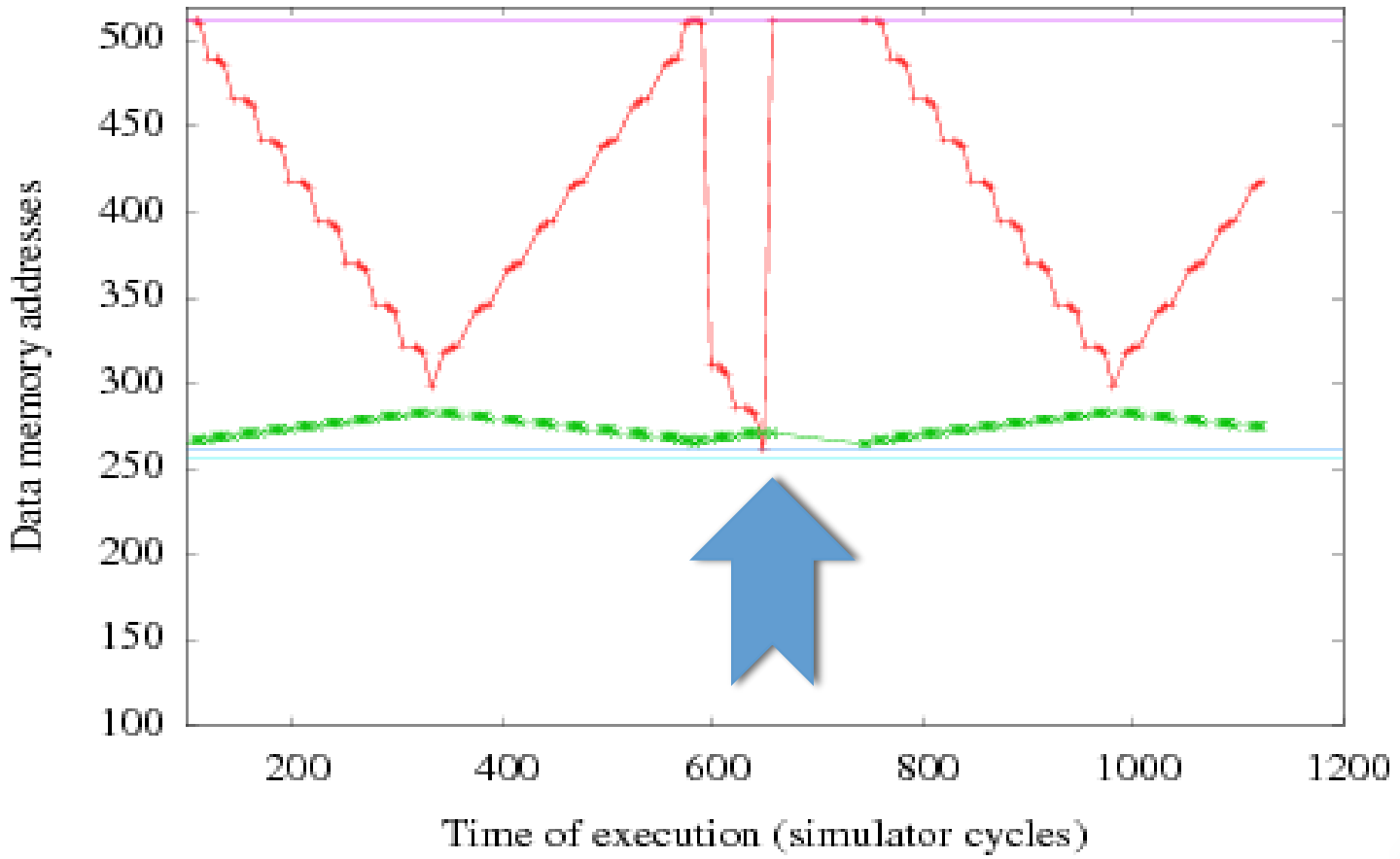
```
…
void factorial_with_bigalloc(){
    volatile uint8_t large[200];
    factorial(8);
}

int main(){
    abssvar=10;
    factorial(8);
    factorial_with_bigalloc();
    return 0;
}
```

Example execution no protection



Example execution with protection



Split stack approach for embedded systems

- The proposed technique
  - no extra memory needs
  - Lightweight hardware modifications
  - Protection against security and reliability problems for MMU-less embedded systems
  - No runtime overhead
  - Needs to be enabled permanently otherwise if an attacker has full control, it can restart the MCU on a modified program and deactivate the SSTACKEN configuration register.
  - Permantent enabling can be done by an irreversible configuration fuse

- Hardware solutions are cheap and effective to increase their security
- Legacy is not as a big problem as in commodity systems

Software-Based Code Attestation

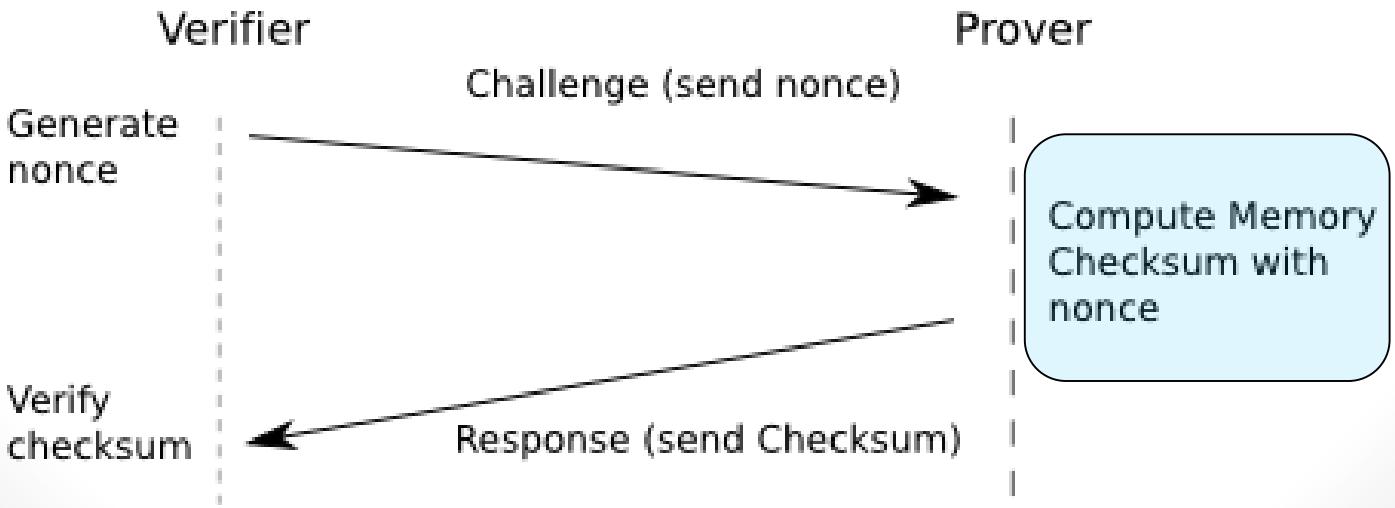
Basic Idea: check whether or not only intended code is present on devices, if yes everything okay, if no attacked

- Objective:
  - Establish trust in a device (without dedicated HW)
  - Detect corrupted/compromised devices
  - Remotely (e.g. from the network)

- Attacks to prevent
  - Malicious modification of the algorithm
  - Execution of attestation on a backup image
  - Replay attacks

Challenge-Response Protocol

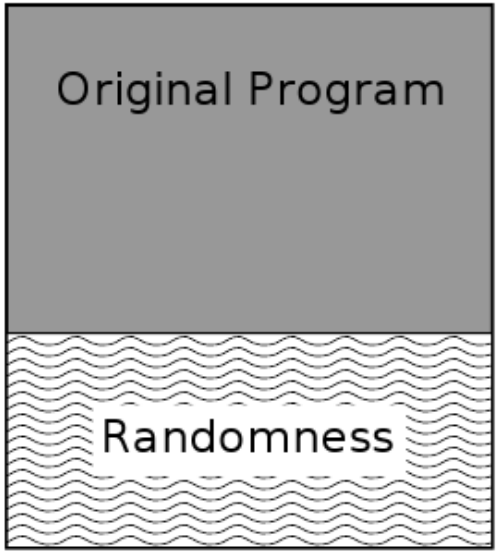
- Common model:
  - The verifier has a copy of the prover's memory
  - Verifier sends a challenge with a nonce to the prover
  - The prover computes a "checksum" of its memory
  - Verifier checks the validity of the checksum



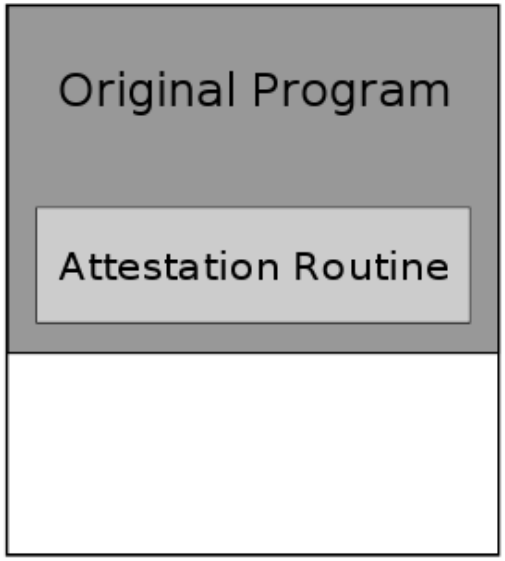
Software-Based Attestation

- 2 main approaches: randomness-based and time-based

- Randomness-based
  - Fill free space with randomness
  - Checksumming all memory
  - No room for malicious code

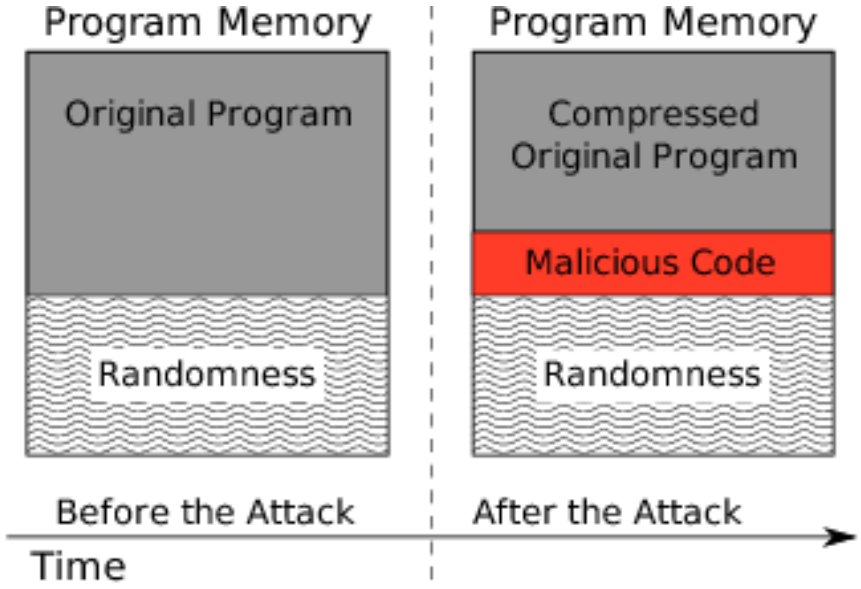


- Time-based
  - Attestation is optimized to be performed in a constant time
  - Code modifications are detected by additional delay



Generic Attack 1 : Randomness…

- Unused space is filled with randomness
- Generic Attack :
  - Compress original code
  - Use freed space to store malicious code
  - On the fly decompression to compute checksum



Generic Attack 1 : Randomness…

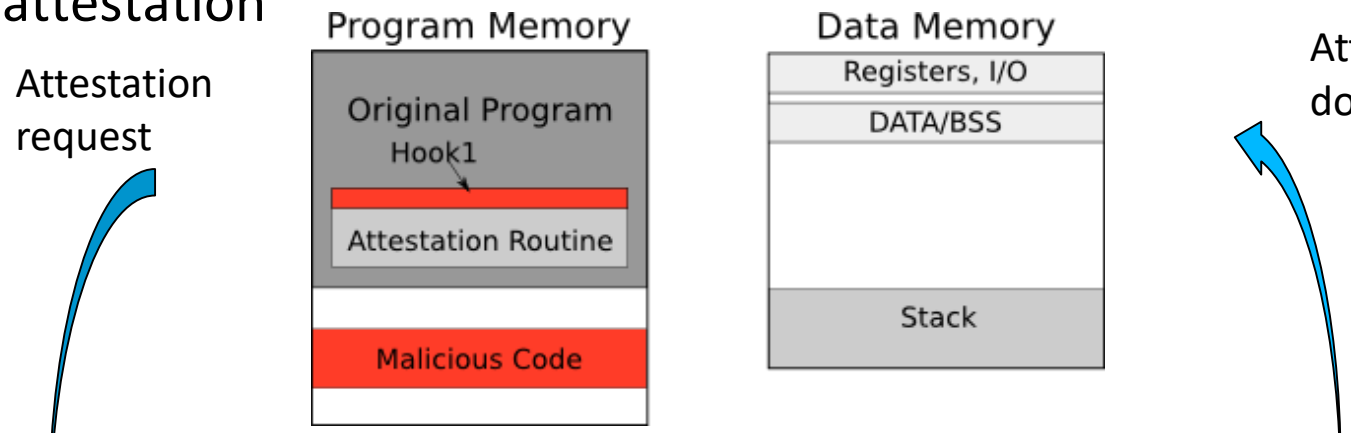
- Attack implemented on a MicaZ
  - Modified canonical Huffman encoding
- On the fly decompression: sequential access
  - No compression: 1 second
  - With attack: 6 seconds
  - Freed space 2220 bytes
- On the fly decompression: random traversal of memory
  - No compression: 145 seconds
  - With attack: 269 seconds (+85%)
  - Freed space: 1252 Bytes
  - Time/memory tradeoffs

Generic Attack 2: Checks PM Only

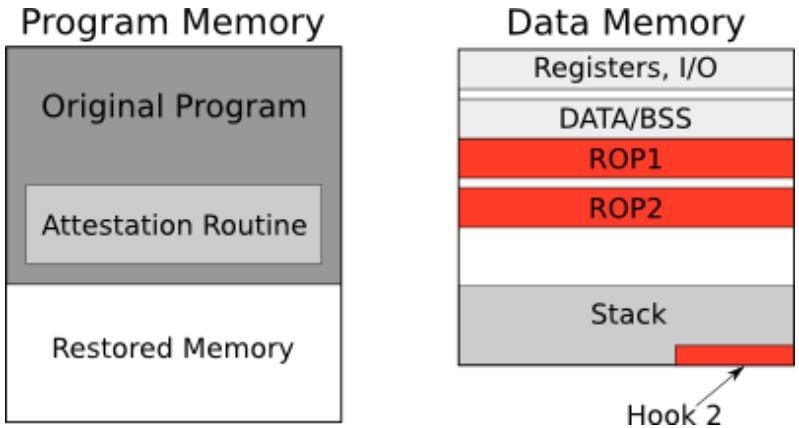
- Attestation is done only on program memory
- Attack: a “rootkit”
  - Hides malicious code in non executable memories
  - Uses return-oriented programming techniques
  - Attestation is performed on program memory only
    - malicious “code” not detected
  - Malicious code restored after attestation

Generic Attack 2: Checks PM Only

- After attestation
- Before attestation



- During attestation

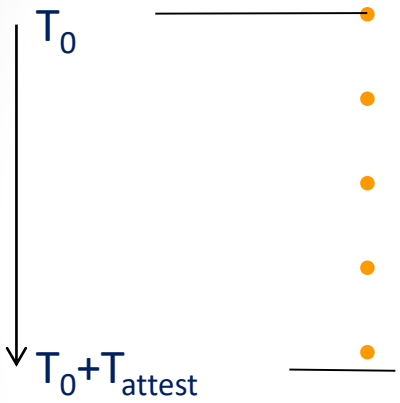


Specific Attacks: SoftWare-based ATTestation: SWATT

- SWATT SoftWare based ATTestation [Seshadri 04]
  - Checksum only the program memory
  - Time-based
  - Timing of sensor response to identify malicious activity

- SWATT relies on:
  - Having the fastest possible implementation of the checksum routine
    - Any changes in delay will be noticed
  - No faster attack exists than expected
  - Ability to make accurate timings (e.g. proximity)

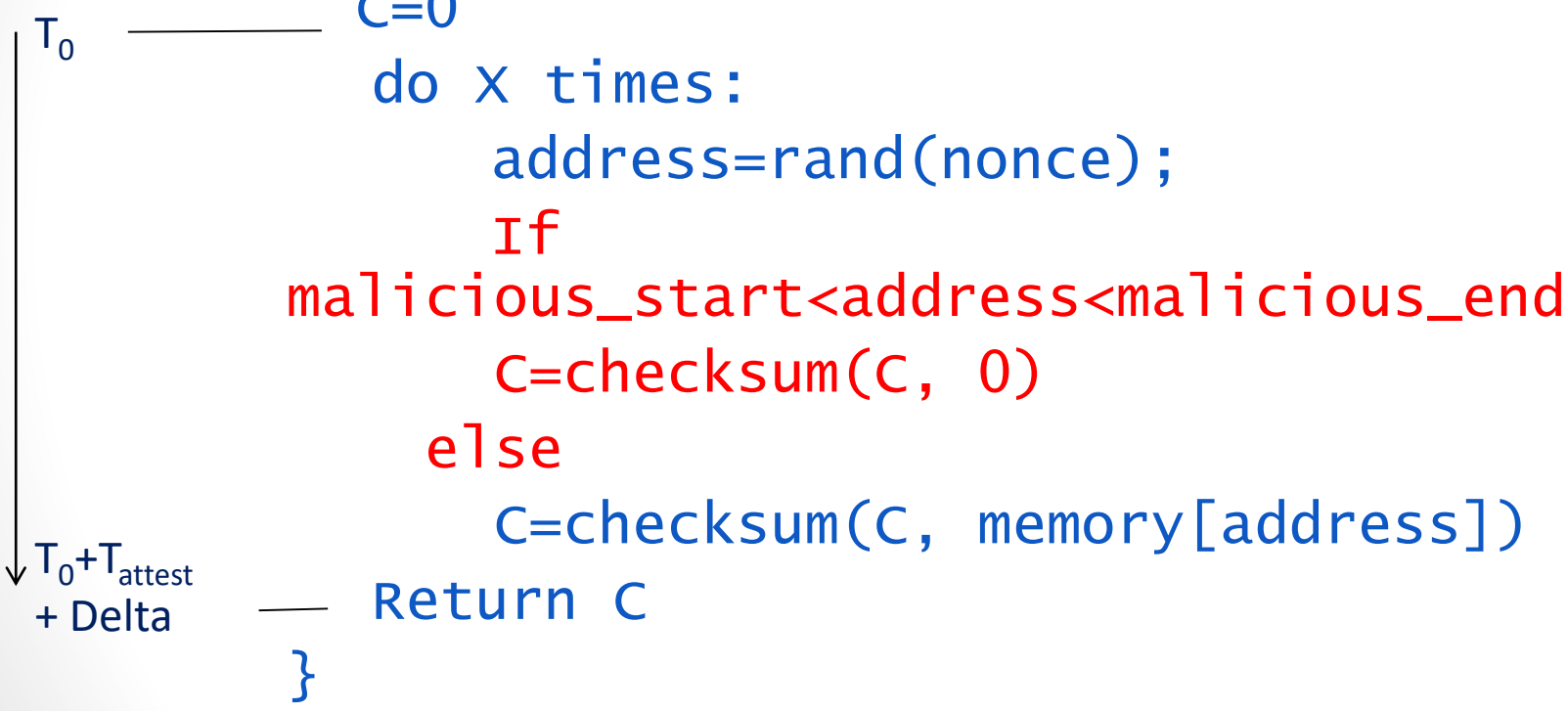
SWATT: Attestation Routine




```
Swatt_Attest (nonce){
  C=0
 do X times:
      address=rand(nonce);
      C=checksum(C, memory[address])
 Return C
}
```

- Main loop
  - executed thousands of times
  - 23 CPU cycles

SWATT : expected attack

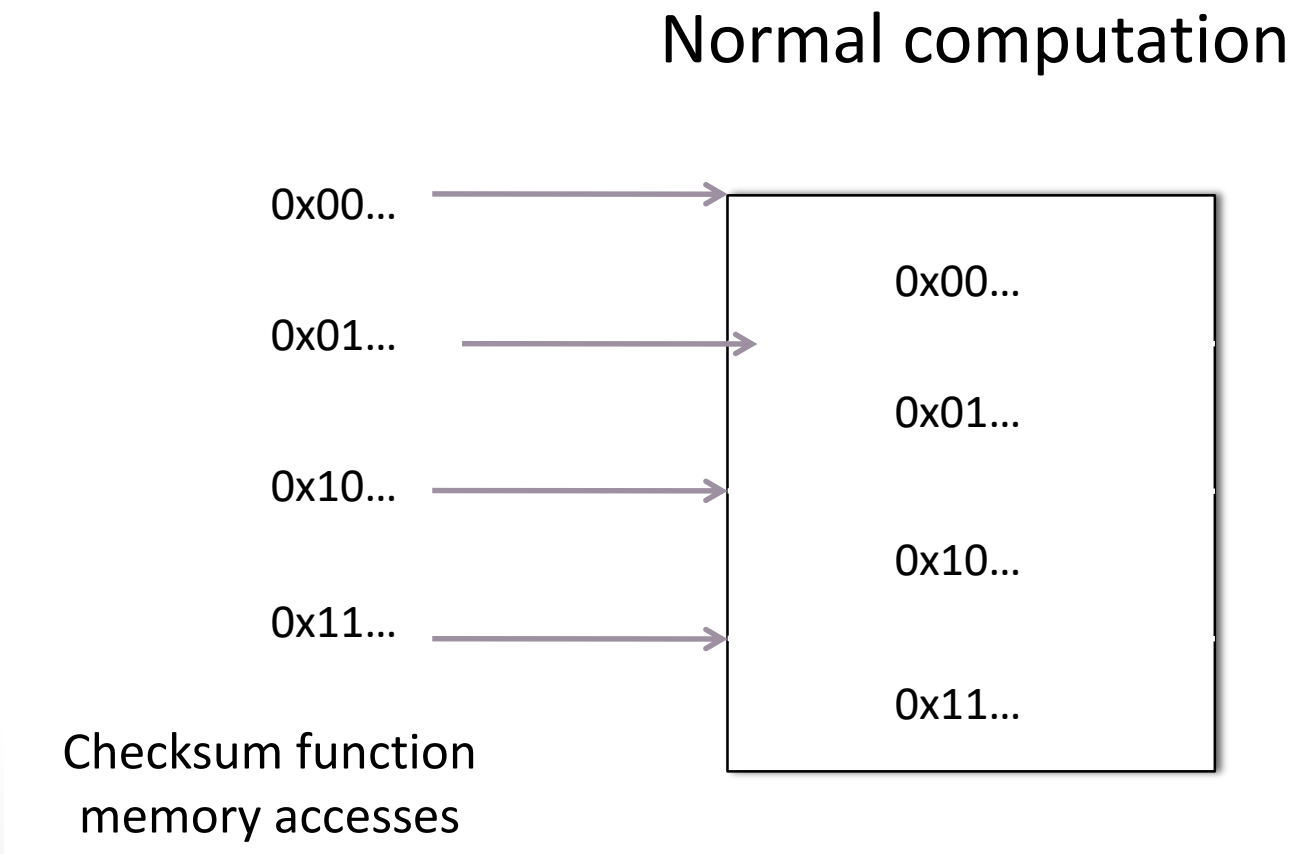


- If Delta >= Best known attack time
  - Attack detected!

SWATT: Memory shadowing attack

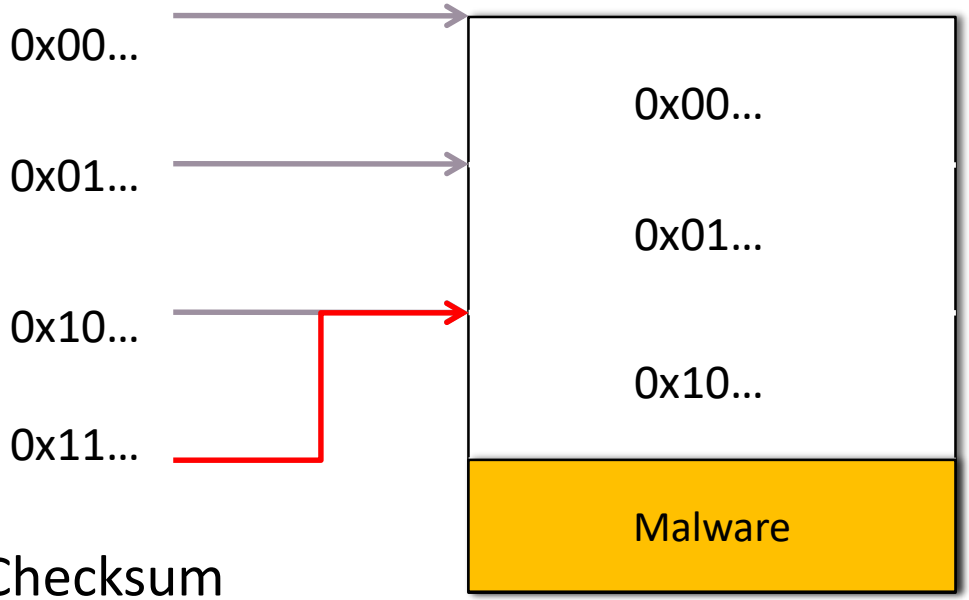
- Memory shadowing attack against SWATT
- Best attack originally expected (13% overhead)
  - Compare and branch
  - 3 cycles
- Shadowing attack by Francillion (7.4% overhead)
  - Conditionally flipping a bit of address
  - 2 cycles
- Assumes that half of memory is empty
  - Application typically 5-50kB while Flash is 128kB

SWATT: Memory shadowing attack



SWATT: Memory shadowing attack

With shadowing



Modified Checksum function memory accesses

Concluding on SWATT

- Reduced the attack from 3 to 2 cycles
  - Is there a faster attack?
  - How much overhead is acceptable?

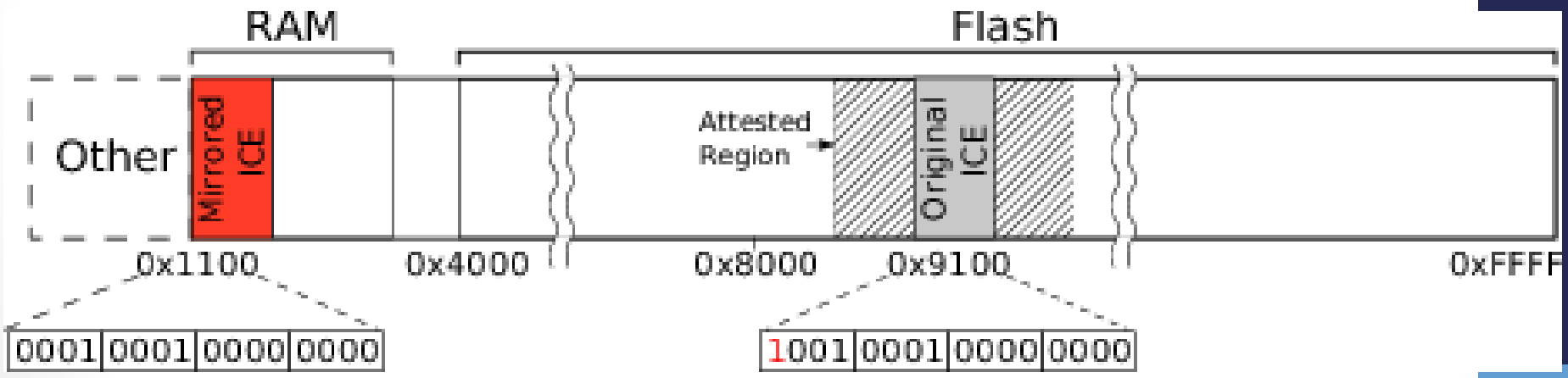
- Rootkit attack is also possible against SWATT
  - Time overhead 0.3%

Specific attacks: Indisputable Code Execution (ICE)

- ICE Indisputable Code Execution (MSP430)
  - Only attest a small region of code
  - then execute attested code.
- As with SWATT:
  - Verifier checks the delay and the checksum result
  - Modification are detected by increased delay
- Checksum is computed using more values of the state (Program Counter, Status Register, etc.)
  - PC included in order to attest the attestation function

Indisputable Code Execution: Attack

- Specific attack
  - Checksum is weak, changes to multiple values cancel each other
  - Checksum function uses " …+ PC Xor Mem[address] +…"
  - To prove "indisputable execution"



- Flip one bit of PC and one bit of all memory regions

Conclusions

- Time based protocols
  - Rely on dedicated checksum functions
  - Designed to be fast, often weak
  - Not portable from device to device (not shown here)
  - Hand optimized in assembly: bugs (not shown here)
  - Getting accurate timings can be a problem if many hops involved

- Memory filling with randomness do not consider:
  - All the memories of the device => Rootkit attack
  - Compression attack

Conclusion

- Requirements to limit possibilities for malicious code
- Needs to attest all memories
- Needs to prevent compression attack
- Needs to rely on strong cryptography
  - Rules out time based schemes

- Software attestation is not easy
  - New attacks presented
  - Would I use software based attestation ? Not for now …

Perspectives

- WSN devices used to get those results
  - applies to many embedded systems that rely on similar MCU
- Interesting fields of applications to explore:
  - Pacemakers
  - Smart Meters

Enforcing the execution of legitimate Software only

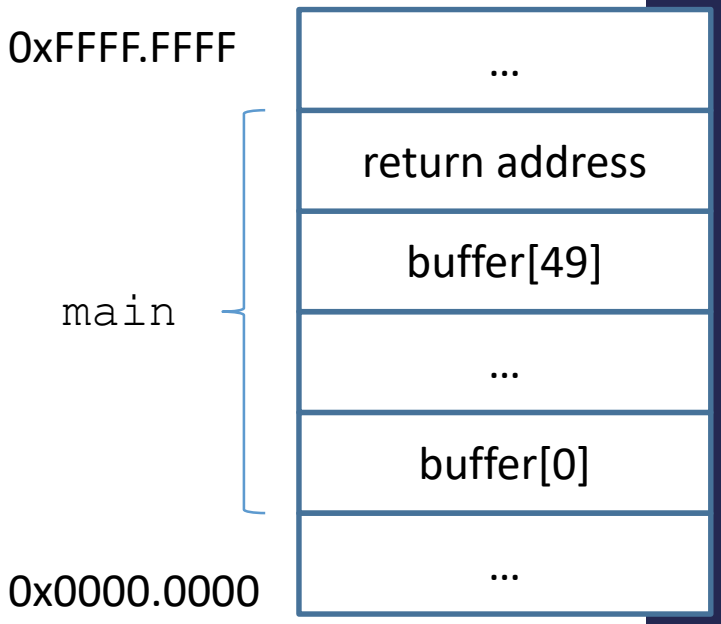
- External program memory
- Misuse of programming interface
- Code injection attacks
- Return-oriented programming attacks (return-into libc attacks)

Threats

Return to libc Attacks (recap)

- Misuse of a buffer overflow to get access to libc
- Example:

```
int main(int argc, char** argv)
{
        char buffer[50];
        strcpy(buffer, argv[1]);
        return 0;
}
```



- Find address of /bin/bash
- Buffer not protected

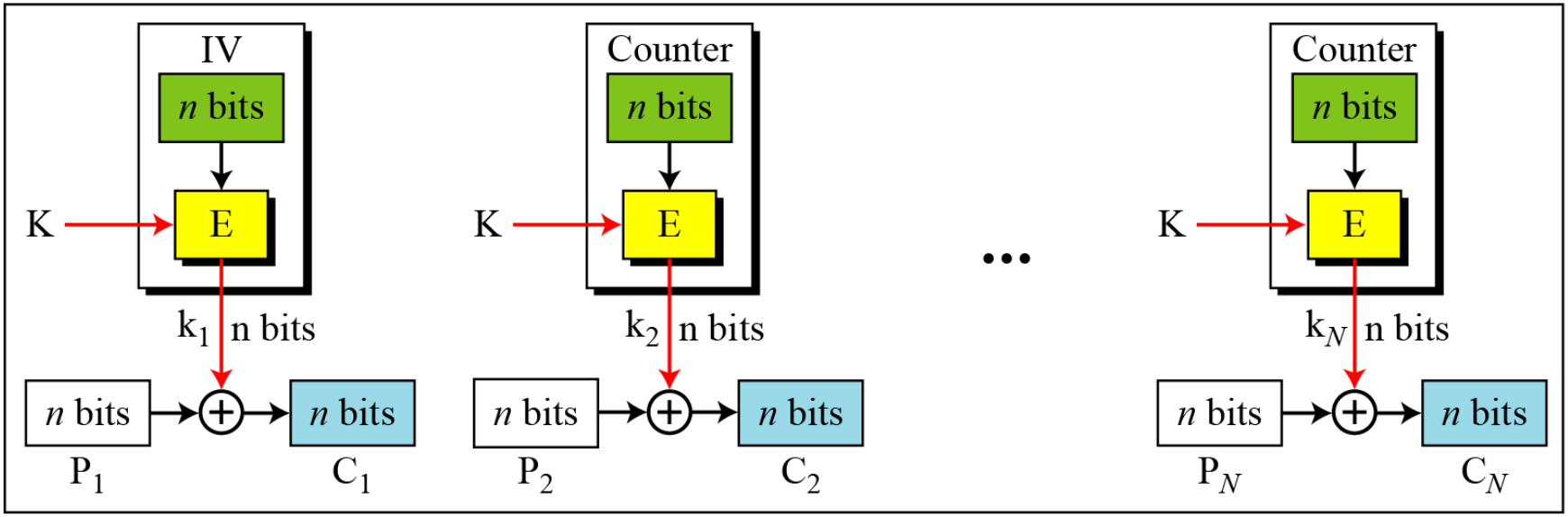
  ⇒ Fill up buffer and add address of bash shell
- Return address is overwritten
- Instead of return to function caller => bash executed
- Reason:
  - Every function can be called from everywhere at any time

Instrinsic Code Attestation: Background CTR mode

E : Encryption
P_i : Plaintext block *i*
K : Secret key

IV: Initialization vector
C_i : Ciphertext block *i*
k_i : Encryption key *i*

The counter is incremented for each block.



Encryption

Idea: Instrinsic Code Attestation

- Instruction depends on previous instruction

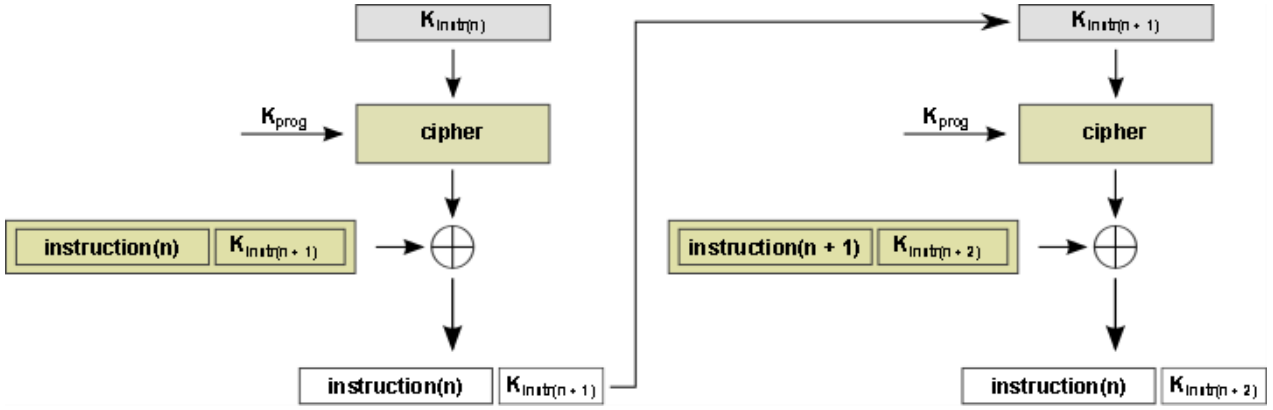
  ⇒Predefined execution order

  ⇒No malicious code can be inserted

- Processor executes encrypted programs

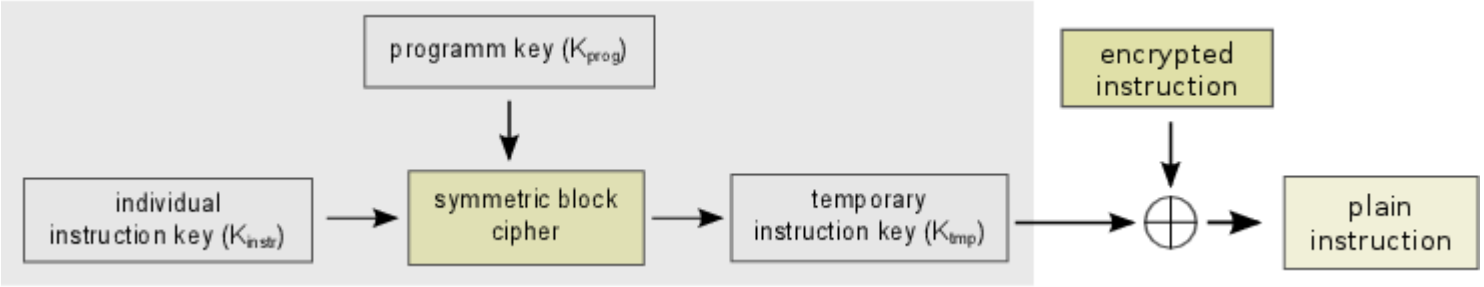
  ⇒ Prevents read out memory

⇒ Called „Instrinsic Code Attestation“ (ICA) with instruction chaining



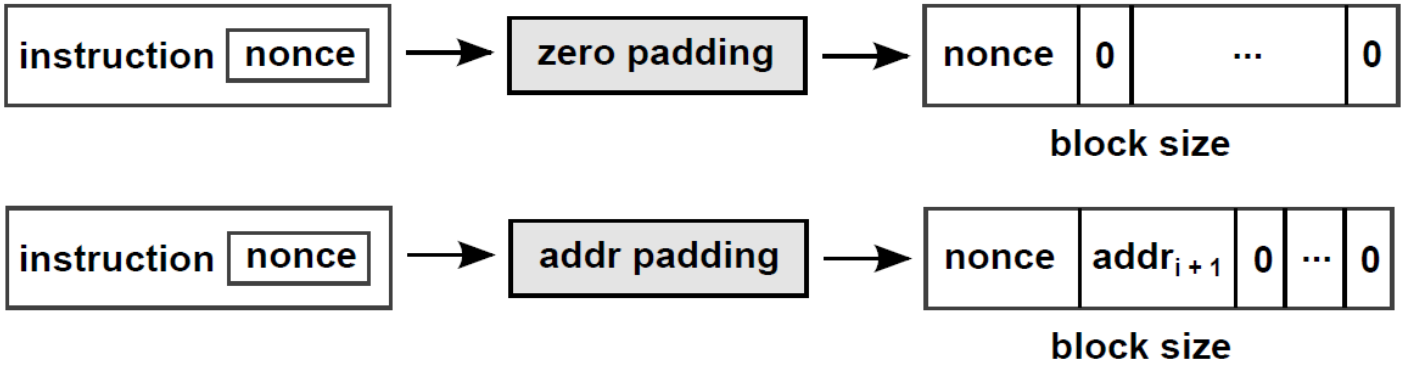
Key / Cipher

- Symmetric block cipher is required in MDU
  - Fast decryption in one clock cycle
- AES is too slow and too large
- PRINCE algorithm is suitable for ICA
  - 64 bit cipher optimized for embedded applications
- Program key is device specific
- Individual instruction key depends on previous instruction

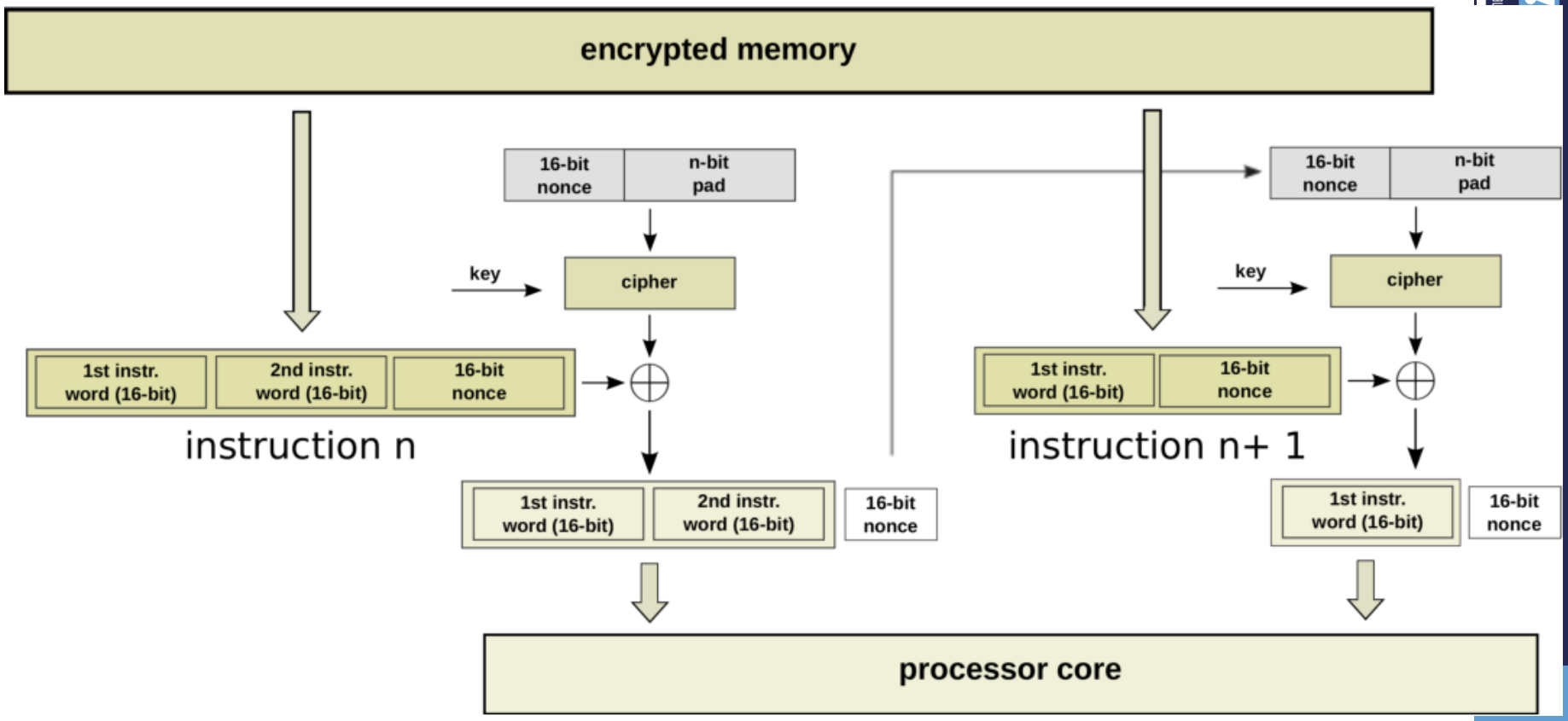


Keys used in ICA

- Keys used in the system
  - Individual Program Key (IPK): kind of master key, permanent
  - Instruction Individual Key (IIK): used ensure chaining of instructions
  - Temporary Instruction Key (TIK): used to decrypt instruction
- Instruction Key padding

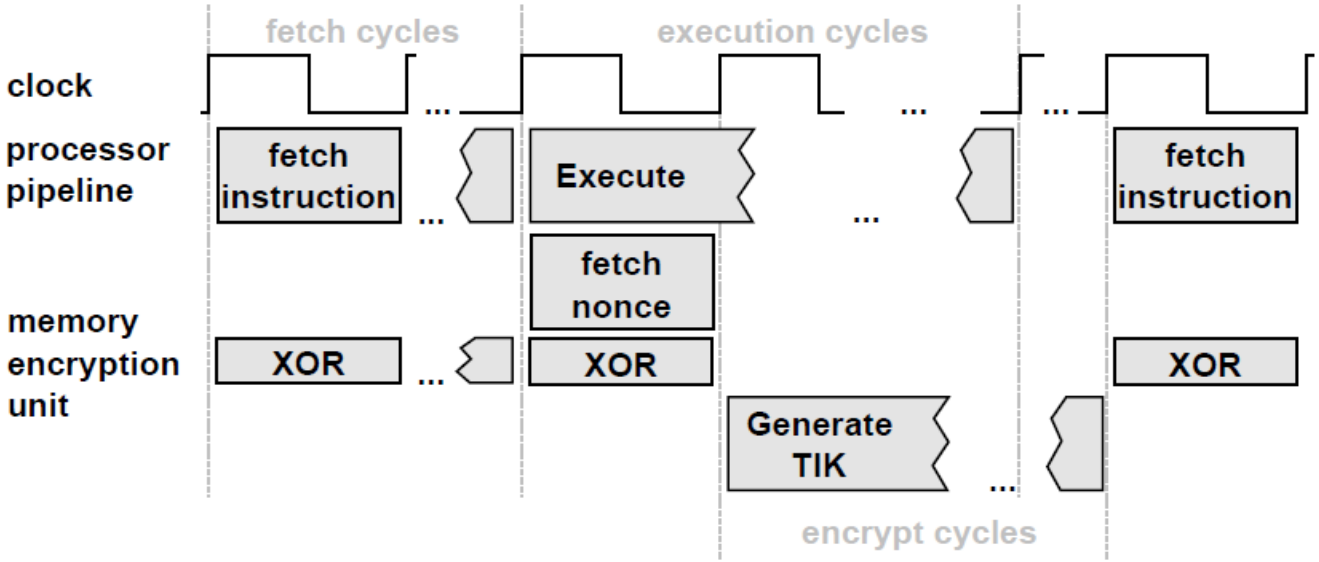


Program Execution



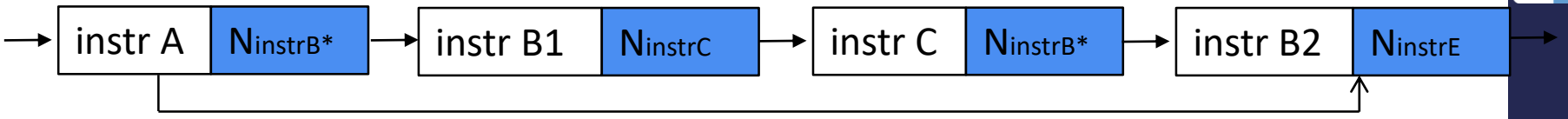
Keys used in ICA

- Generating the TIK

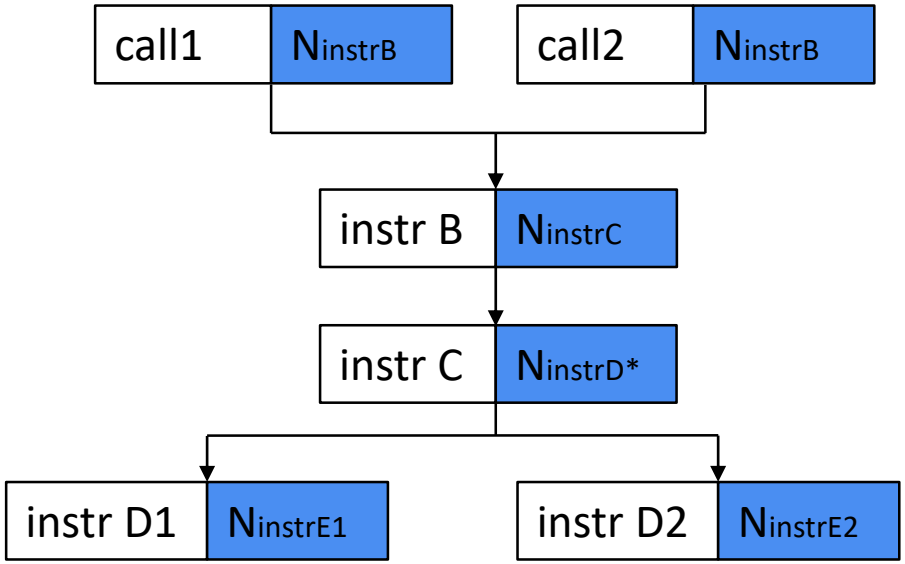


Special Cases in Program Flow

- Conditional jump



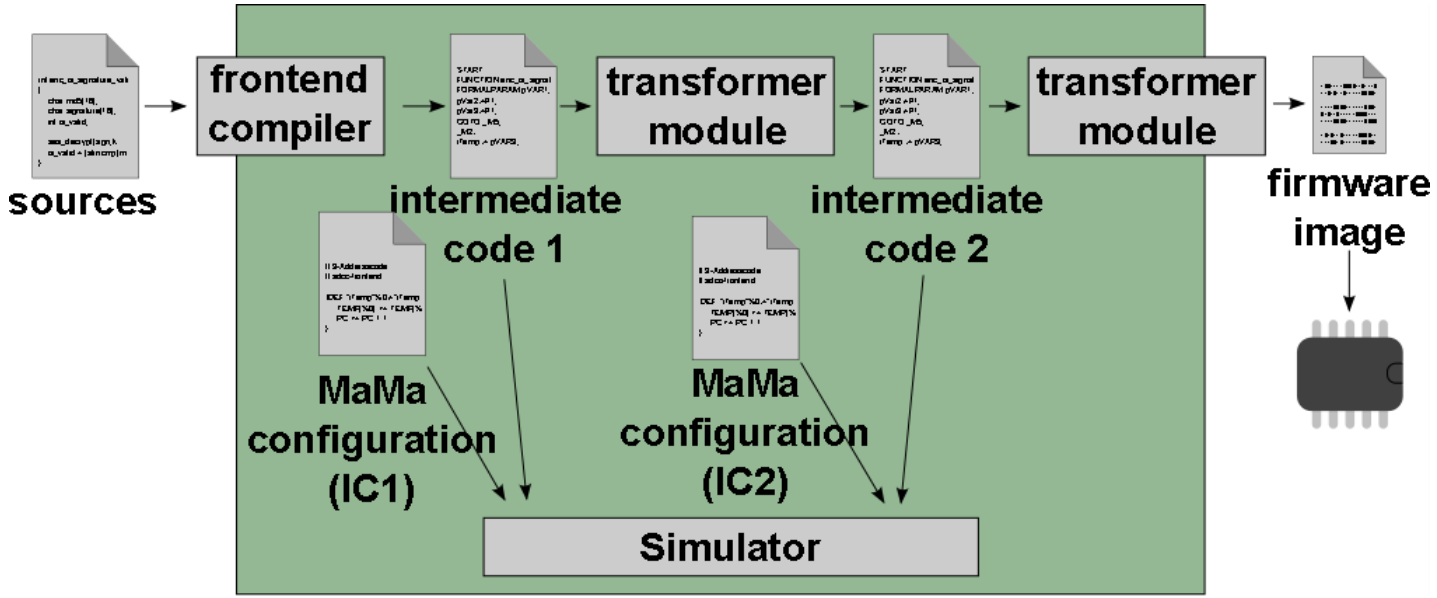
- Function calls



- Interrupt
  - Stack for nonces

Adaption in Software Flow (Compiler)

- Modification of binary is required:
  - Add nonce
  - Handling special cases
- GNU compiler can be extend, but high effort
- CoMet tool supports any frontend compiler and transforms to any intermediate code



Adaption of Program

Program (Compiler Output)

Addr	Instruction (bin)	Instruction
4000	12 53	inc r2
4002	fe 3f	jmp \$-2

Modified Program – Nonce added

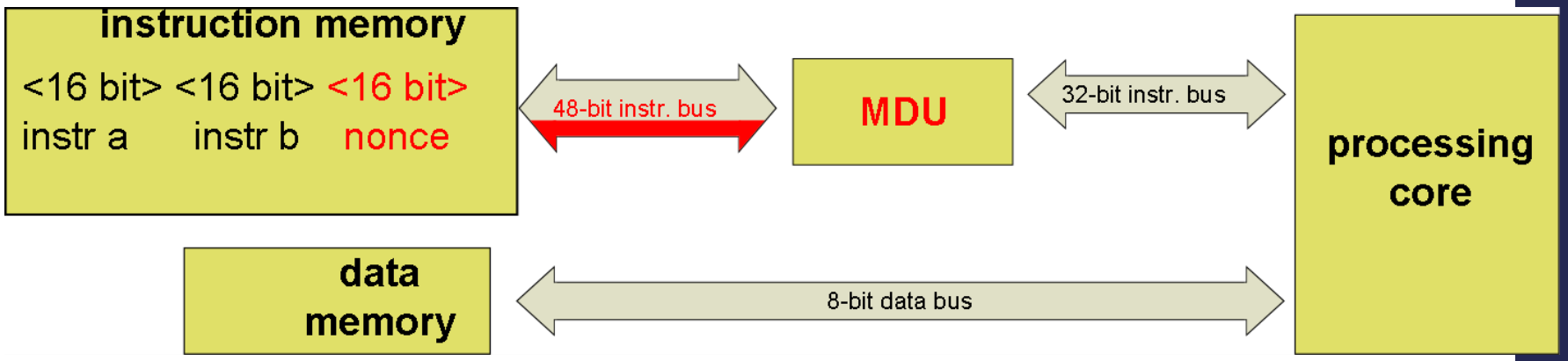
Addr	Instruction (bin)	Nonce	Instruction
…	…	00 00	…
4000	12 53	00 01	inc r2
4002	fe 3f	00 02	jmp \$-2

Encrypted Program

Addr	Instruction (bin) -enc	Nonce - enc	Instruction
…	…	e8 3a	…
4000	45 28	d9 88	inc r2
4002	f7 32	41 a2	jmp \$-2

Hardware Integration of MDU into tinyVLIW8

- tinyVLIW8: RISC archictecture with uniform instruction format



- openMSP430: Von Neumann architecture
- Integration is identical to RISC architecture

⇒ Instruction length from 1 to 4 byte – effort for software adaption is increased

Design size and speed estimation

Decryption in two steps:

1. Running block decryption
   - Logical path has to fufil timing requirements of main system clock
2. Encrypted instruction XOR key => decrypted instruction
   - XOR operation is very fast
   - Influence on time is very low

Softcore	Logical cells
TinyVLIW8	1162
openMSP430	2841
LEON2	9299
MDU (Skeleton without cipher)	126
MDU with PRINCE	2001

Security Analysis

The security of the ICA approach is based on the following assertions:

- A1 Physical and hardware-based attacks on the MDU are beyond the adversary's capabilities.
- A2 The program key can be accessed only from within the MDU. This is guaranteed by the absence of physical lines to read the key outside.
- A3 The MDU cannot be bypassed since it decouples the instruction memory from the instruction decoder. All instructions must pass the MDU.
- A4 The TIK calculated by the memory decryption unit (MDU) cannot be forged. Since the TIK is the result of a strong block cipher with an adequate security level.
- A5 The nonce cannot be replaced by a user defined value. The hardware guarantees that the nonce is directly read from the encrypted instruction memory.
- A6 An instruction can be only decrypted with the correct nonce. The nonce and the instruction address are the initialization vector of the CTR block cipher.
- A7 The program key update is forbidden or protected by a strong authentication scheme.
- A8 Any erroneous decryption results in an unpredictable program behavior or leads to a hardware reset.
- A9 The normal execution of an encrypted program should leak no information about the program key and the encrypted nonces.

Security Analysis: Remaining risks

- key collisions, brute-force attacks, and attacks based on ROP.
- Key Collisions:
  - commodity 16-bit MCUs have an address space up to 22-bit, so TIK collisions cannot be avoided. On an architecture with 22-bit address space each TIK may be used up to 100 times.
  - But as an attacker does not know the IPK it cannot qualify the correct set of TIKs. So, from the perspective of security the reduced number of TIKs and their multiple use is harmless. The probability of guessing a precise nonce of an instruction remains 2^{-16}.

Security Analysis: Remaining risks

- Cipher Instruction Search (CIS) attack:
  - **brute force attack** on the enciphered machine instructions and then observing the CPU reaction. The adversary presents a large number of guessed encrypted machine instructions to the CPU to construct an enciphered program to gain more information or to provide clear text access to the instruction memory. For a CIS attack the target device must be connected to a programming device. We must assume that the device provides access to all processor registers except the MDU internal registers.
  - Depending on the speed of the programming device an attack can be done quite fast. At an MCU clock speed of 20MHz a shot of a single instruction needs only a few milliseconds. So brute forcing all 216 alternatives takes only a few seconds. Furthermore, the brute force strategy can be applied to each instruction in the same way with the same effort.
  - The success of a (CIS) attack can be significantly reduced on systems with larger instructions. Furthermore, adding the instruction address to the nonce pad prevents a copy of guessed program code and makes the reuse of an enciphered program code, which was constructed in RAM, infeasible.

Security Analysis: Remaining risks

- Multiple return points
  - Instruction chaining uses the nonce that was encrypted together with the previous instruction for decrypting the next instruction in a CTR wise fashion. Hence, for sequential code that does not include any branches, a unique nonce is used for encryption of each instruction. This uniqueness assures that only the legitimate previous instruction can be the predecessor of the current instruction.
  - Since we use the first n bits of a block cipher output as the TIK, the probability that two given nonces propose the same TIK is approximately 2^{-n}.
  - Branches => a nonce will be used multiple times, thus allowing an instruction to have multiple successors that can be decrypted with this nonce. This introduces the possibility of undesired modifications in the program flow: multiple instructions sharing a nonce are able to jump to each others
  - BUT: the number of instruction that might be jumped at is significantly reduced to the number of two.
  - Second risk: legitimate jumps that might be taken when they are actually not allowed: the return instruction of a function might have multiple successors corresponding to multiple calling instructions. Although a jump to all these successors is legitimate, only one of these jumps should be taken. The same obviously holds true for conditional jumps where both jumps are valid while only one of them should be taken
  - BUT: in both cases the attack vector is significantly reduced.

Conclusion

- Proof of concept for intrinsic code attestation has been shown
- Execution of encrypted instructions prevents different types of attacks
- Reusing of authorized instructions is not possible – code in a chain
- Area overhead depends on processor type and encryption algorithm
- Performance penalty one clock cycle
- Straightforward integration
- Adaptable and scalable design
- Glueless integration into compiler tool chain possible
- Approach verified in simulation and onto FPGA
- Future work:
  - Applying the approach on blocks instead of instruction
  - Side channel analysis

HW based code attestation for embedded devices: SMART!

The goals:

- Design minimal hardware to support a dynamic root of trust on low-end embedded devices
  - Limited hardware modifications
- Fast: no interruption of normal operation
- Provably secure

SMART!

How:

- Use a small ROM memory to store SMART code
- Implement secure key storage on the MCU
- Only the code in ROM can access this key
- Prevent leakage of the key to the rest of the system
- The ROM computes the MAC of a target piece of code, then (possibly) jumps to it

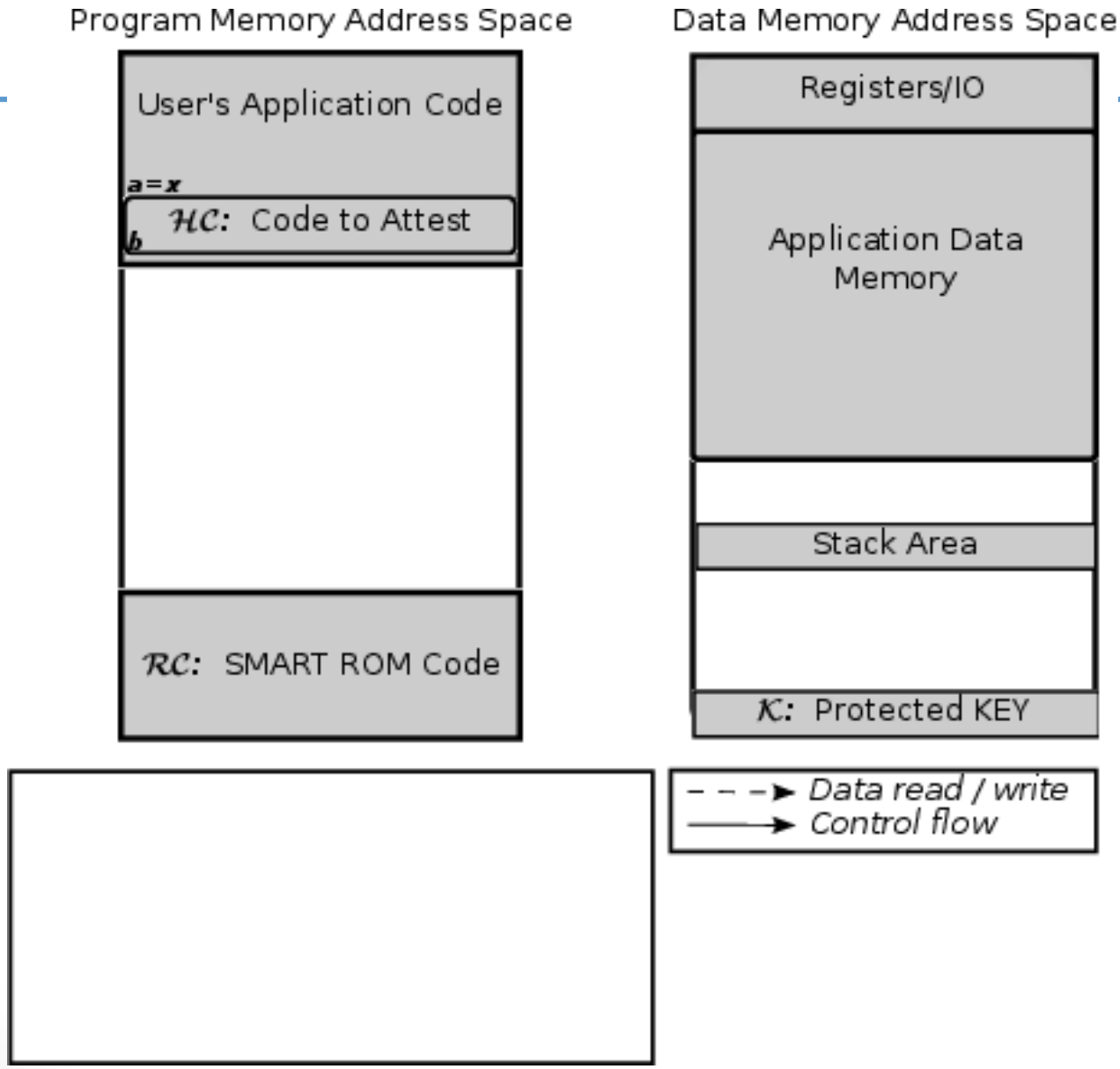
SMART

- Attestation Read-Only Memory
  - region of memory in ROM inside the MCU
  - only this code segment is granted access the attestation key by the MCU
  - Performs an HMAC
- Secure Key Storage
  - Memory region inside the CPU can be accessed only from SMART code in ROM
- MCU Access Controls
  - Control access to attestation key and prevent non-SMART code from accessing it

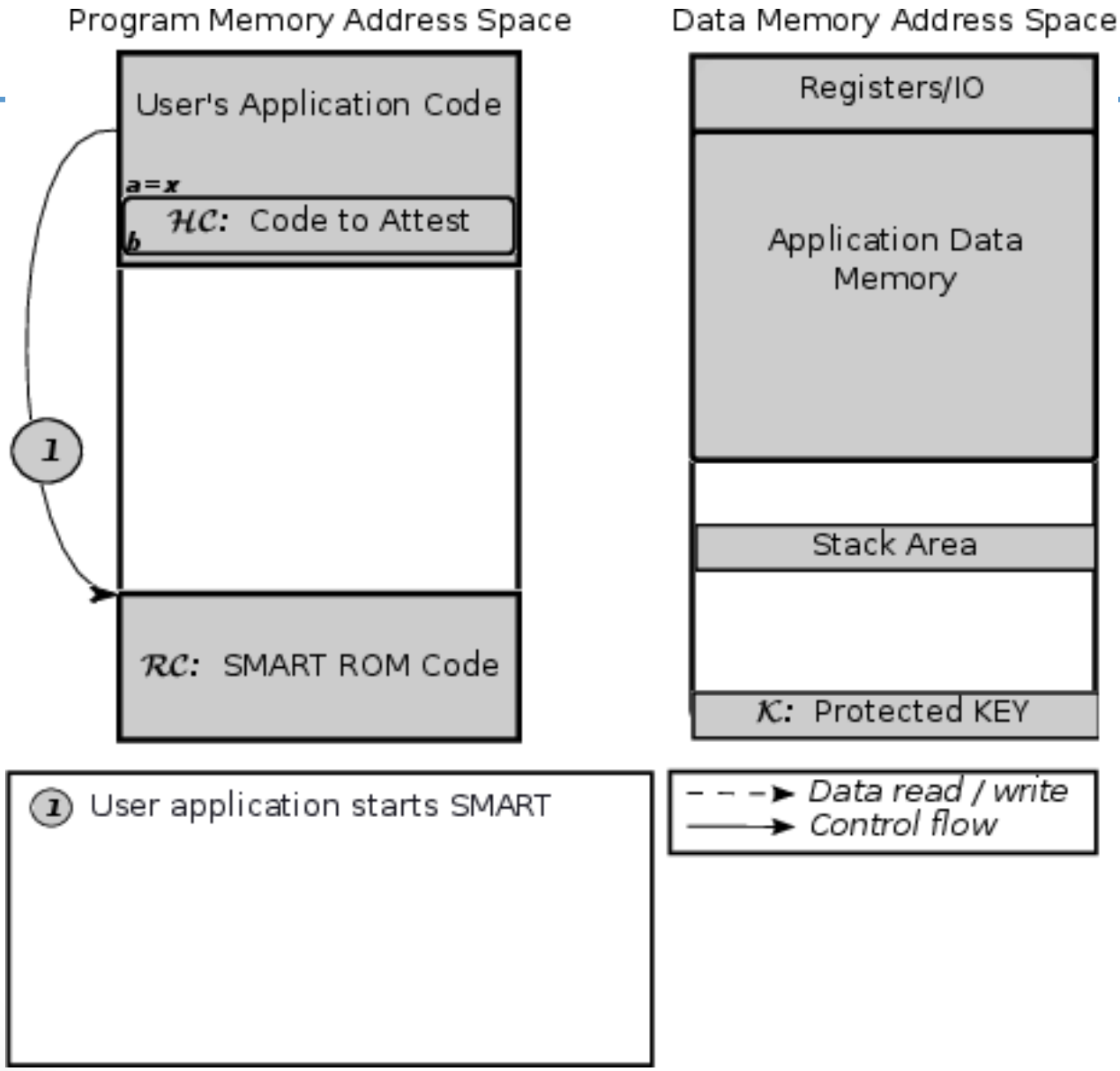
SMART: The protocol

- Verifier → Prover: (nonce, a, b, exec, ...)
- Prover invokes SMART(nonce, a, b, exec, …)
- SMART computes HMAC and then jumps to exec
  - MAC over parameters, data
- Prover sends the result of the HMAC to Verifier
- Verifier can check the HMAC and decide whether to trust the Prover

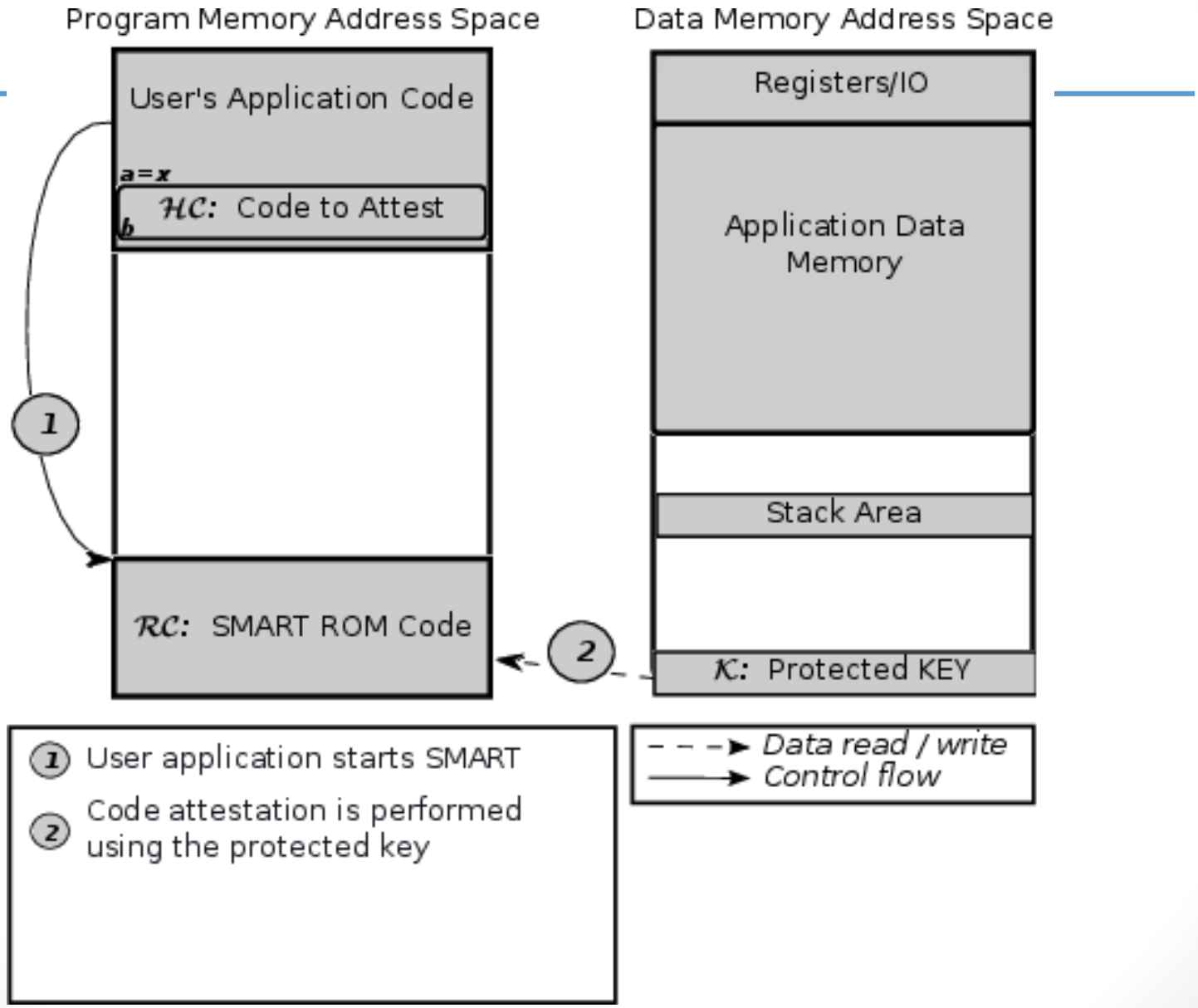
SMART Overview



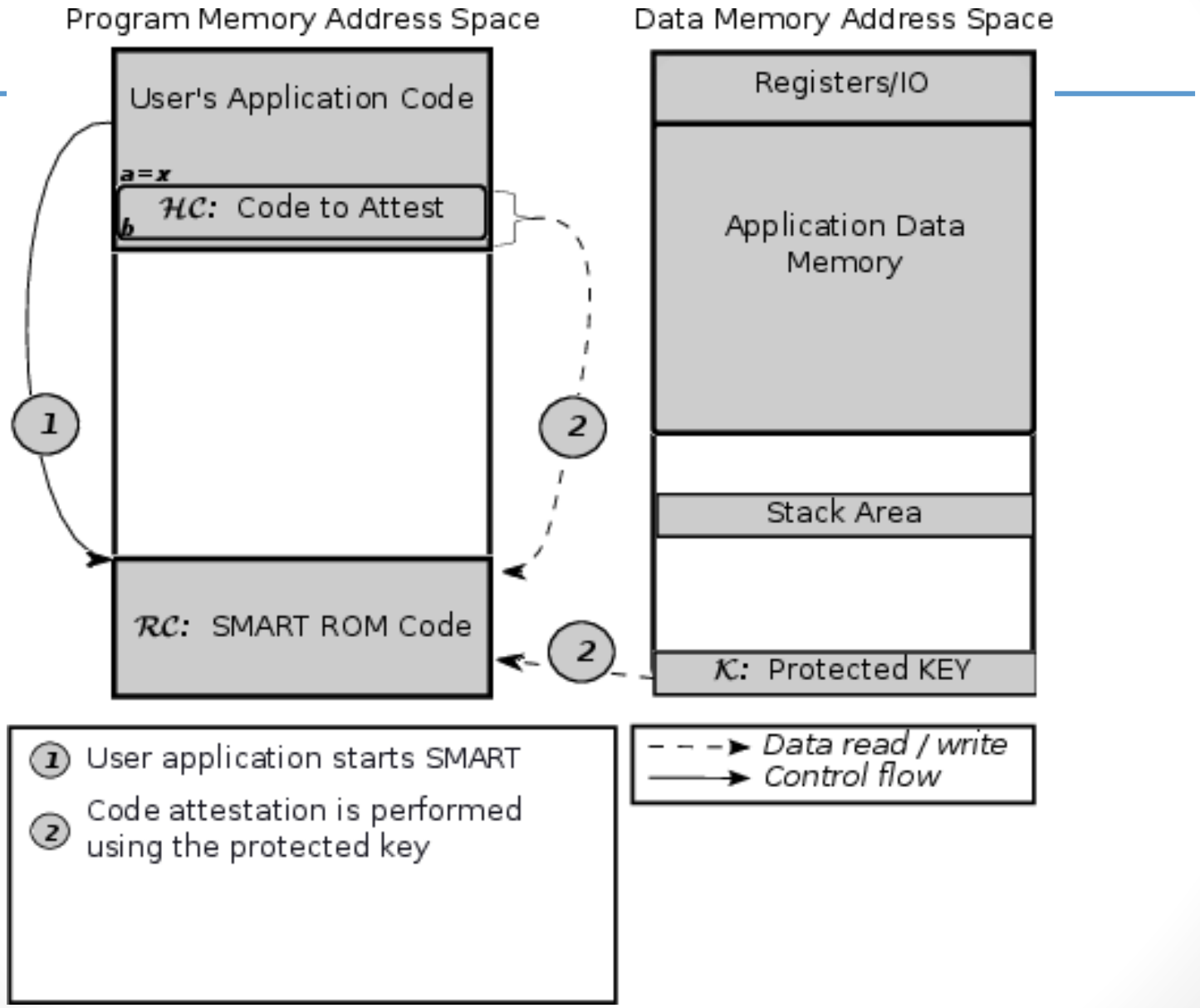
SMART Overview



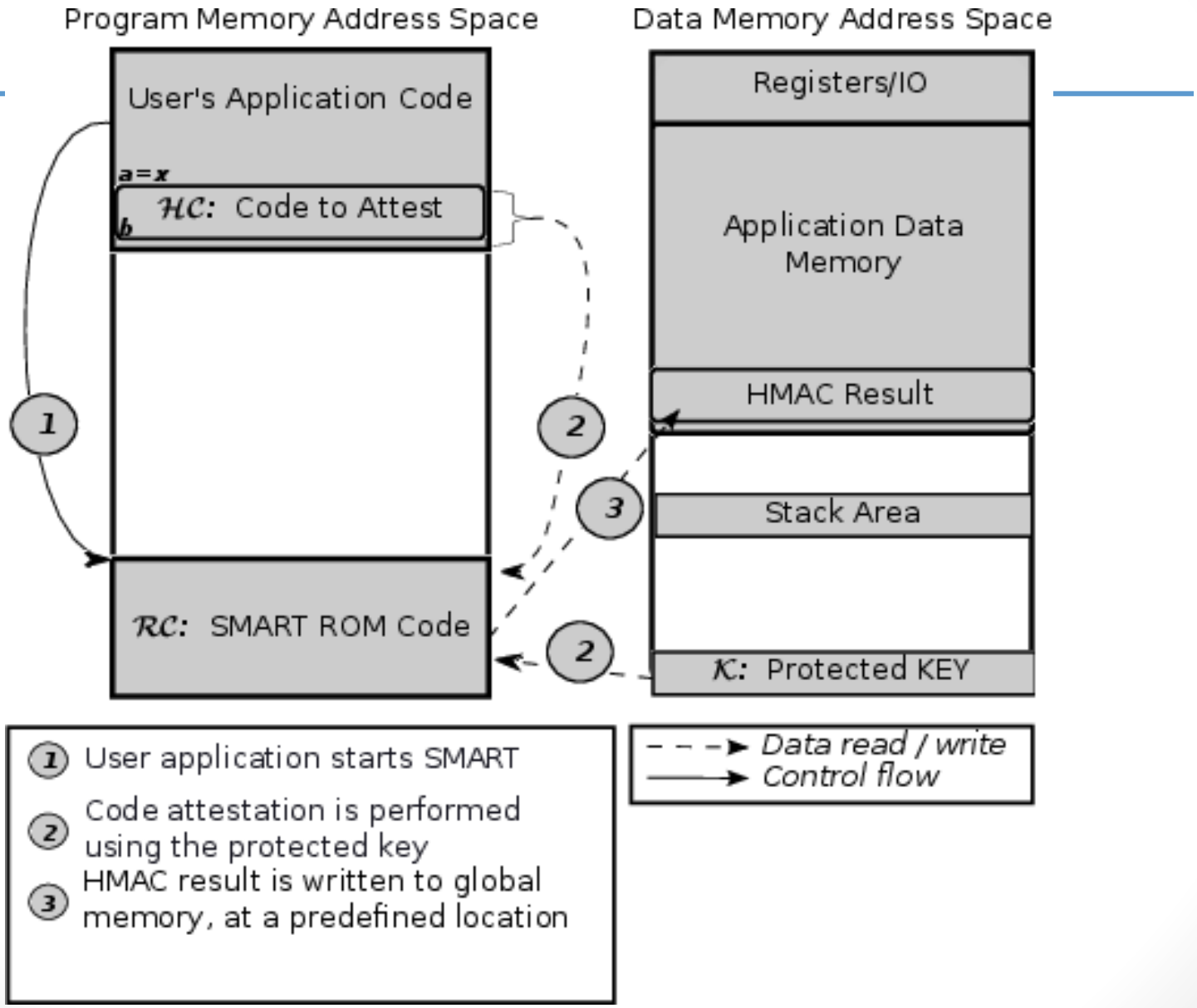
SMART Overview



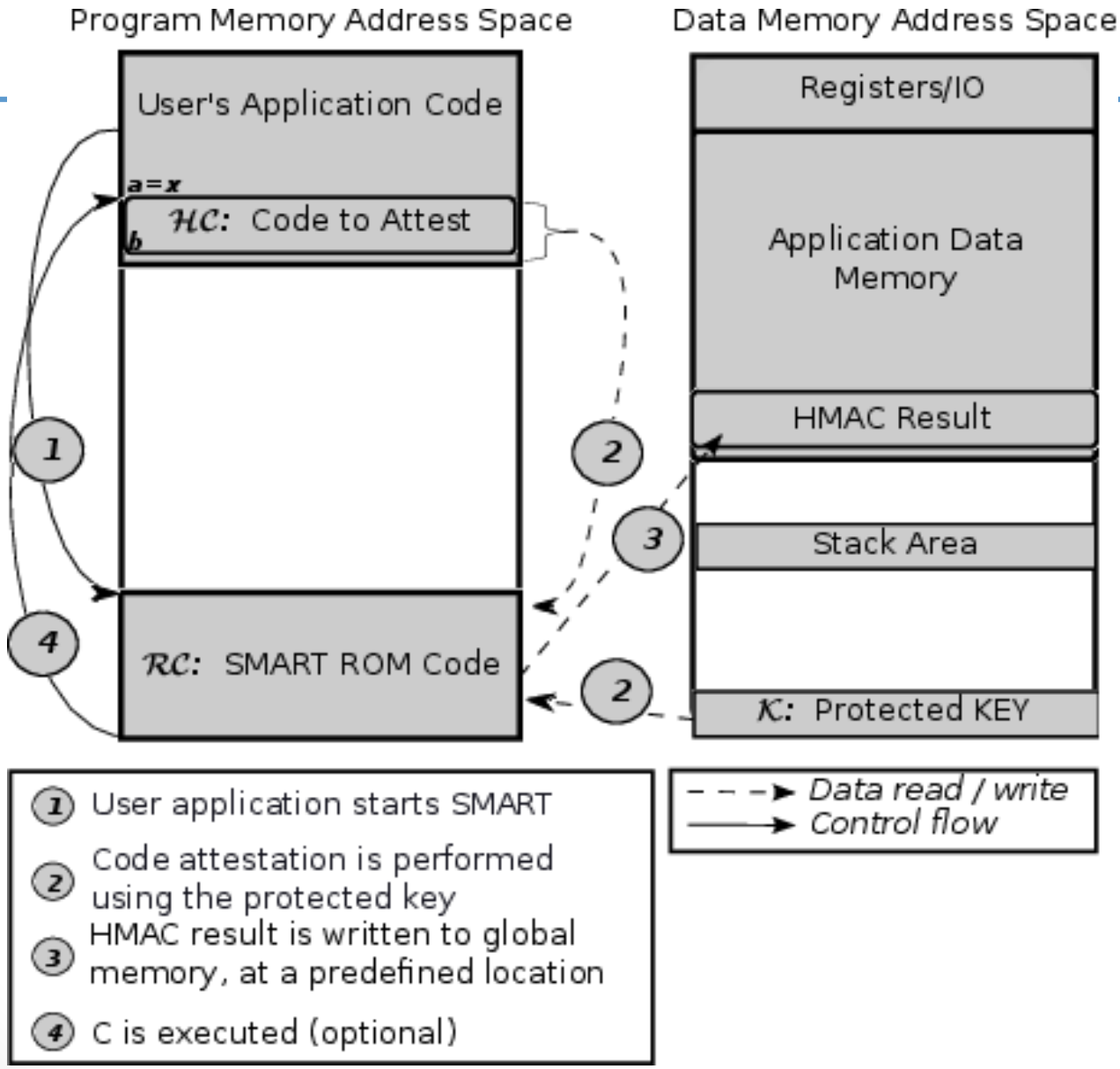
SMART Overview



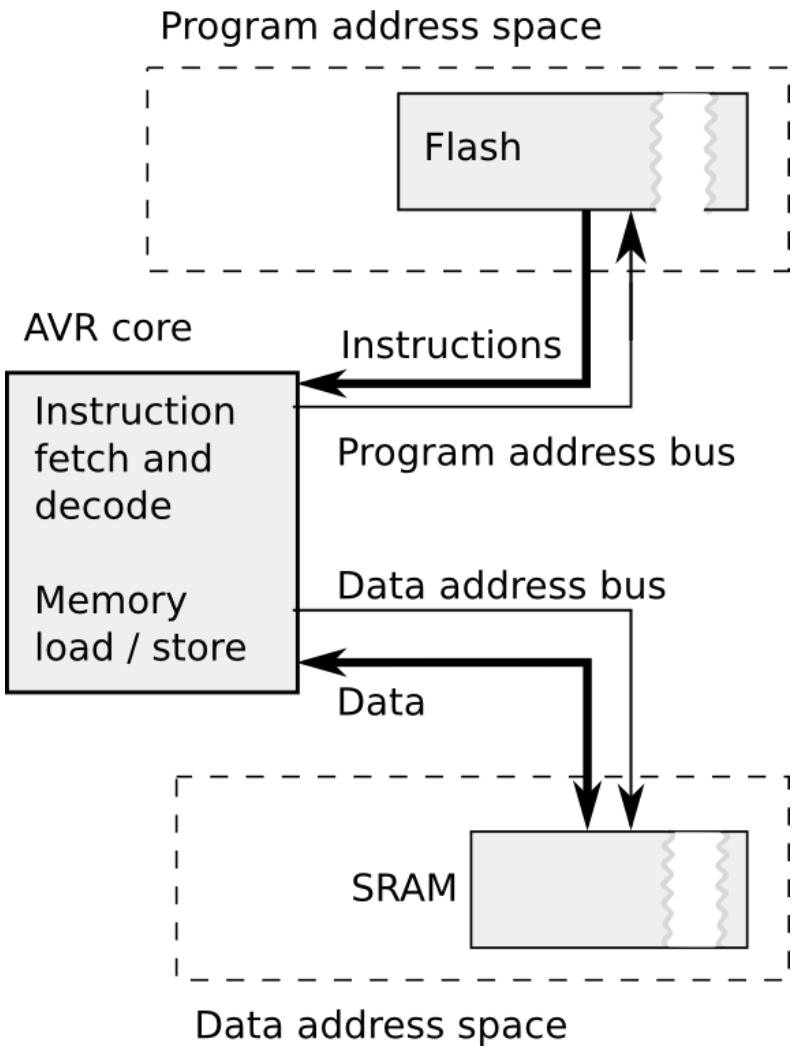
SMART Overview



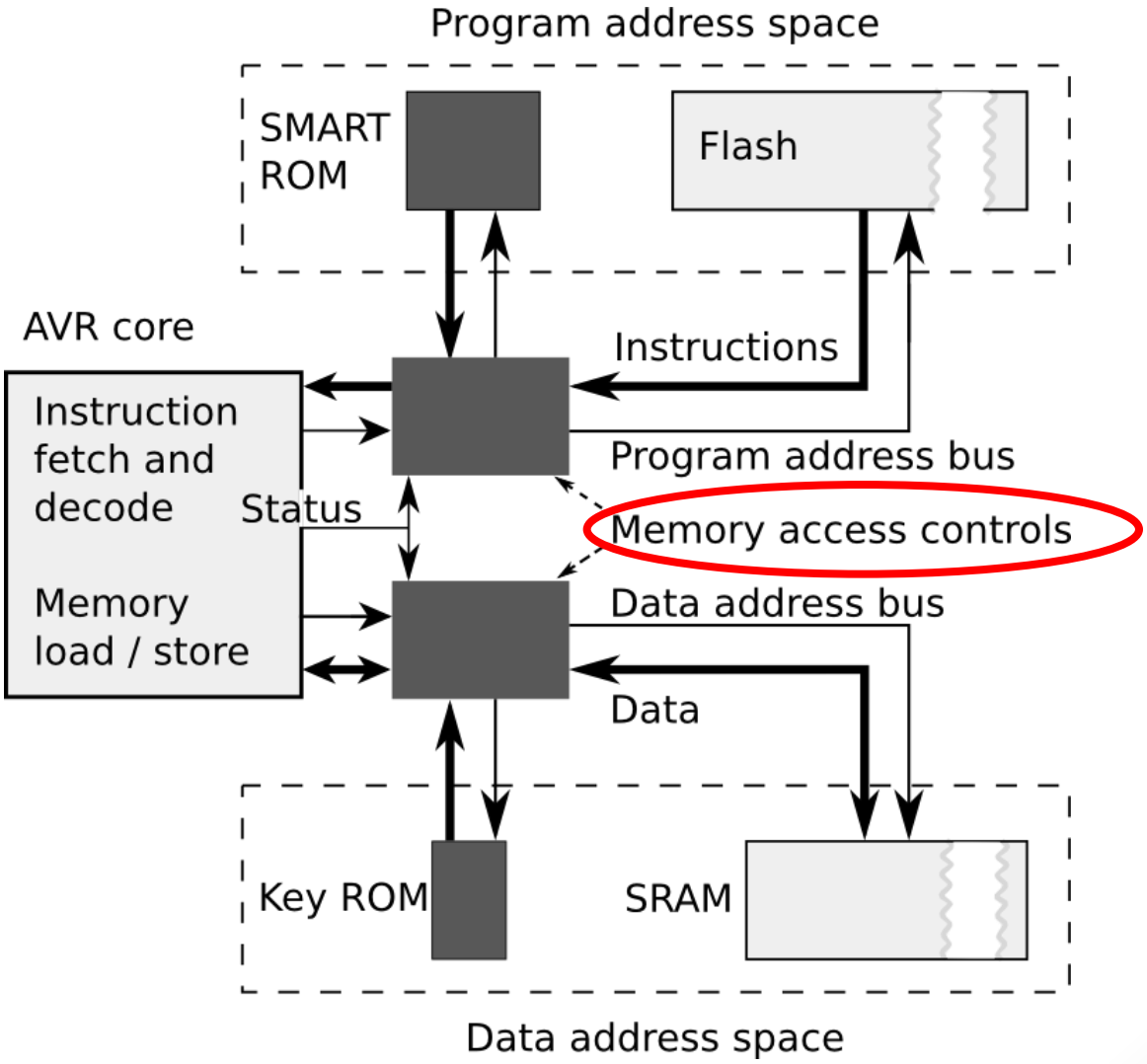
SMART Overview



SMART on the AVR



SMART on the AVR



SMART: Security Properties

- Prover Authentication
  - External Verification
- Verifier knows the content of [a,b] on the Prover
- Guaranteed Execution
  - Dynamic root of trust
- Verifier knows that the prover executed the target code
- Key Protection
- The key is never leaked to code outside ROM

SMART: Implementation

- All the modifications are confined to the memory controller
- No new instruction added to the instruction set
- Invoking SMART requires only a function call
- No additional programming interface needed
- Implementation done on AVR and MSP-430
  - two very different architectures

HW Overhead

Component		Original Size in kGE	Changed Size in kGE	Ratio
AVR MCU		103	113	10%
Core		11.3	11.6	2.6%
Sram	4 kB	26,6	26.6	0%
Flash	32 kB	65	65	0%
ROM	6 kB	-	10.3	-
MSP430 MCU		128	141	10%
Core		7.6	8.3	9.2%
Sram	10 kB	55.4	55.4	0%
Flash	32 kB	65	65	0%
ROM	4 kB	-	12.7	-

To Conclude

- Embedded devices currently used have little detection means against runtime attacks
- Attacks have been shown that subverted the code
- SMART provides a simple and efficient way to establish a dynamic root of trust and guaranteed execution
- Implementation on 2 MCU cores
  - Was actually manufactured on silicon on small quantities
    - NXP 90nm
    - IHP 250 nm

Protection Means

- Avoiding exploiting Stack buffer overflows
  - Software based e.g. canaries: Applicable in most cases but with performance penalty
  - Hardware based e.g. split stack: Merely a theoretical thing as it requires changing processor architectures

- Verification of software integrity
  - Remote Code attestation
    - Pure software: Applicable in most cases but „simple“ to trick out
    - Hardware based: Merely a theoretical thing as it requires changing processor architectures

  - Ensuring software / control flow integrity
    - Pure software
    - Hardware based: Merely a theoretical thing as it requires changing processor architectures

Detection of compromised nodes

- Code attestation
  - Verification of memory of node
  - Software or hardware for verification (TCG)
  - Hardware cost issue
  - Most pure software based schemes have been proven to be insecure
- Secure misbehavior detection
  - Voting schemes
  - Misuse of voting by malicious nodes
  - Limitation of vote per node (sybill attack?)

Detecting Misbehaviour by external observation

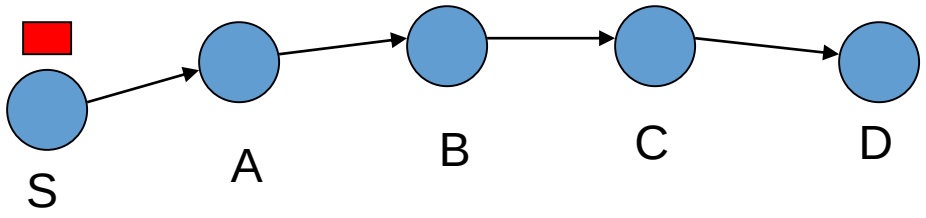
- ***Watchdog and Pathrater, Mitigating routing misbehavior in mobile Ad hoc networks," Mobcom'00.***
- , "Performance Analysis of the Confidant Protocol," Buchegger & Boudec Mobihoc'02
- "Stimulating cooperation in self-organizing mobile ad hoc networks," Buttyan and Hubaux, MONET 2002.
- ***RANBAR: RANSAC-based resilient aggregation in sensor networks ....Buttyan, Schaffer , Vajda, 2006***

Watchdog and Pathrater

- Misbehaving nodes
  - Selfish, malicious, overloaded, broken
- Basic idea: identify misbehaving nodes and avoid them in routing.

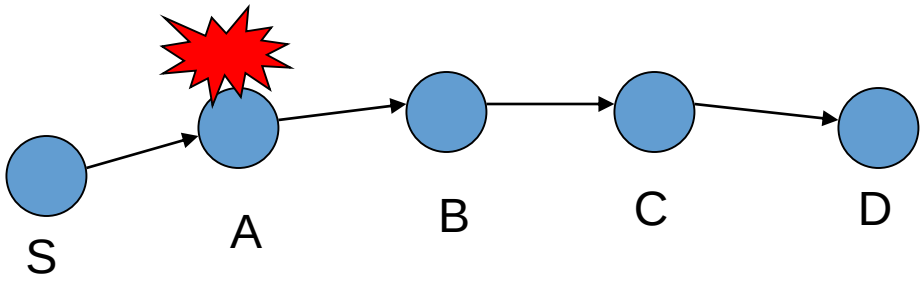
Watchdog

- A scheme to identify misbehaving nodes
- On top of dynamic source routing
- Monitors next node's transmission
- Tallies its misbehaviors
- Reports its misbehaving status when tally reaches a threshold



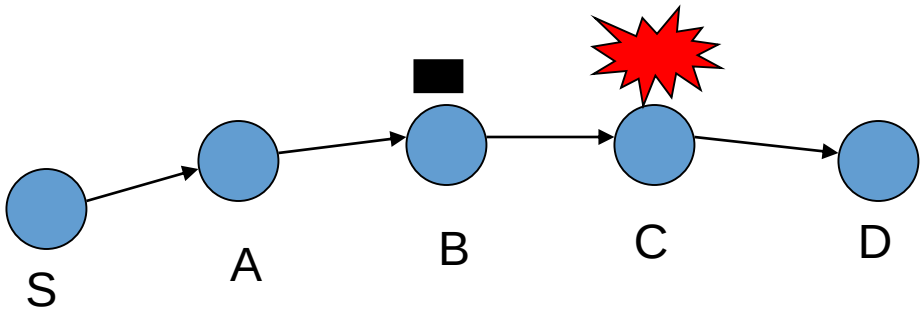
Watchdog's Weakness (1)

- Ambiguous collision: while A is monitoring B's forwarding, it hears a collision.
- Question: has B forwarded the packet?



Watchdog's Weakness (2)

- Receiver collision: a packet forwarded by B may collide at C.
- Problem: a selfish B may choose to forward any packet only once?

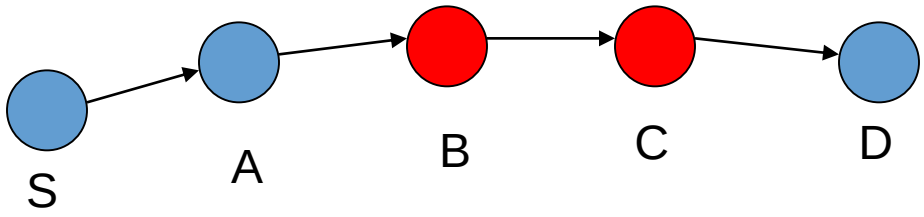


Watchdog's Weakness (3)

- Partial dropping: the watchdog reports misbehavior only if it reaches a threshold.
- Problem: a selfish node may choose to drop packets at a "safe" rate?

Watchdog's Weakness (4)

- Collusion: two or more nodes collude to cheat.
- Example: C always drops packets, but B does not report it.

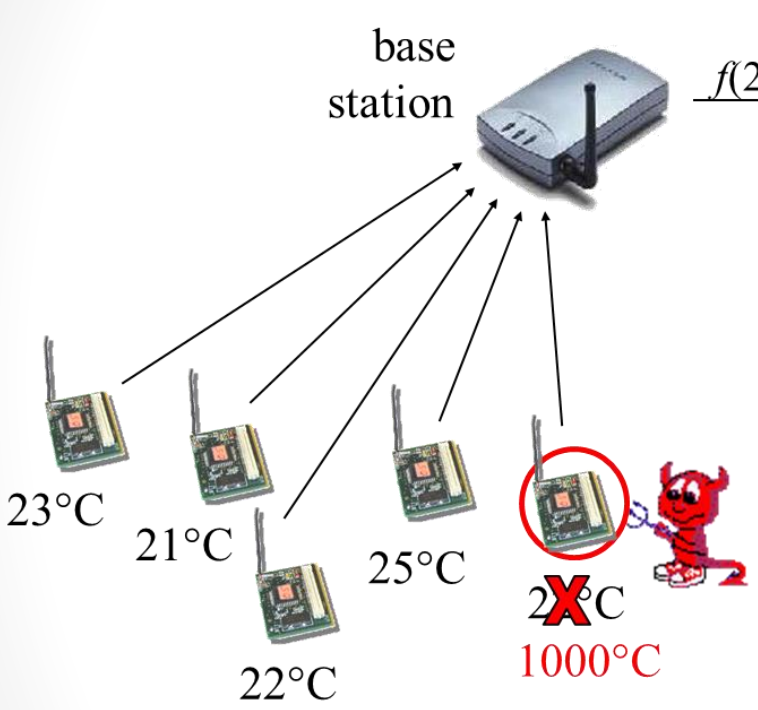


Attacks & countermeasures: Service Integrity

- Stealth attack: acceptance of false data
  - Reporting fictitious/biased data
  - Effect increased in combination with sybill attack
  - DoS to block correct results

- Dissemination of false timing information to de-synchronize the nodes

Countermeasure: RANBAR



Computing the average temperature + an attack

where $f(x_1,\ldots,x_n) = (x_1+\ldots+x_n)/n$

The RANBAR algorithm filters out the compomised elements by

1. taking a small random set of the sample
2. fitting a model on that set
3. selecting the elements that are not consistent with the model
4. if the number of selected elements is low (i.e., the model describes the sample well), they are dropped and RANBAR ends in aggregating the remained elements
5. otherwise, all the intermediate results are rejected and the algorithm starts again

Questions ?