Compilation de protections logicielles contre les attaques par injections de fautes

Advanced secu – M2 CySeC 2025

Damien Couroussé – CEA List Grenoble
damien.courousse@cea.fr

bio

damien.courousse@cea.fr

Ingénieur de recherche HDR – CEA-List

- Logiciel embarqué
- Compilateurs
- Microarchitecture des processeurs
- Sécurité matérielle :
  - attaques par canaux auxiliaires,
  - par injections de fautes
- Méthodes formelles pour les analyses de sécurité

Internships 2025

Thématique sécurité : vérification de la robustesse d’un circuit aux injections de fautes
Design of Fault Injection Models Within Pre-silicon Security Methodologies
https://www.emploi.cea.fr □ 37787

Thématique sécurité : support de calcul pour le chiffrement homomorphe sur RISC-V
FPGA Prototyping of Fully Homomorphic Encryption on RISC-V Microprocessors
https://www.emploi.cea.fr □ 37898

Thématique cryptographie : implémentation matérielle d’algorithme postquantiques
Towards Efficient and Secure Keccak Acceleration: Optimizing Masked Hardware Against Side-Channel Attacks
https://www.emploi.cea.fr □ 37905

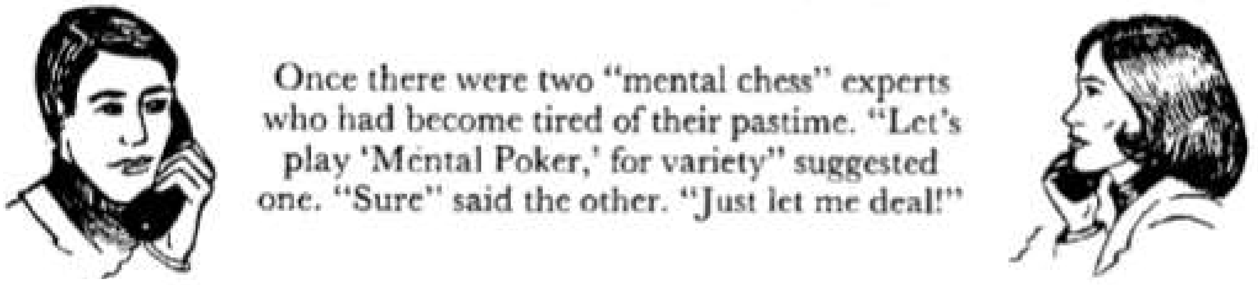
Thématique architecture : design mémoires, architecture des processeurs et de la hiérarchie mémoire
Hybrid DRAM: Next gen ferroelectric memory for “normally-off / instant-on” embedded systems

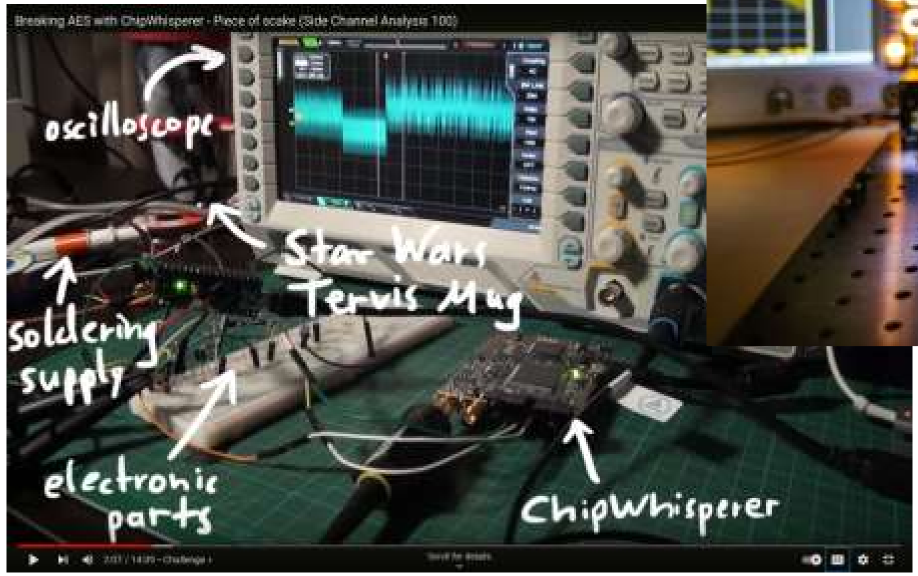
agenda

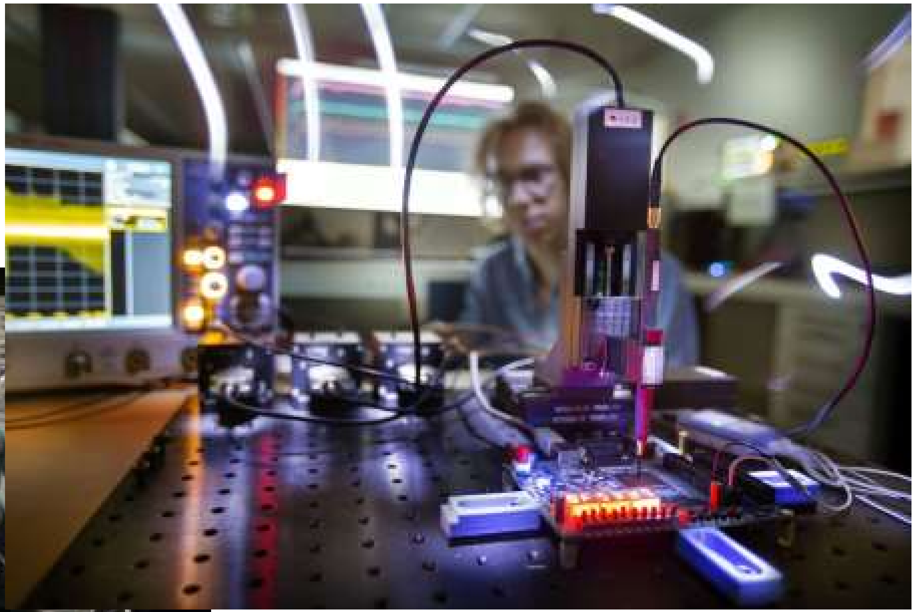
1. **(re-)Introduction to hardware security**
2. **Workflow production for numerical systems and the compiler**
   - A standard compiler doesn't know about security… illustrated
3. **Securing compilers, illustration on simple cases**
4. **Robustness analysis against fault injection attacks**
   - Fault modelling
   - Two main approaches: simulation, formal verification.
5. **Introduction to the hands-on session: the verifyPin case study**

Hardware Security Matters

Cybersecurity: a challenge for the information society


Once there were two "mental chess" experts who had become tired of their pastime. "Let's play 'Mental Poker,' for variety" suggested one. "Sure" said the other. "Just let me deal!"

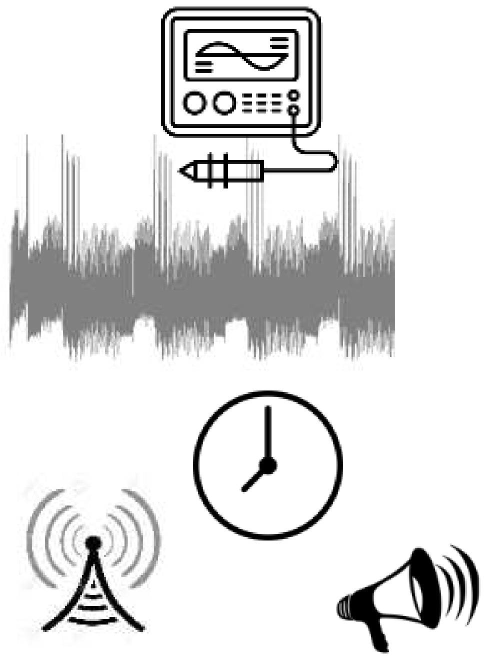

Breaking AES with ChipWhisperer - Piece of scake (Side Channel Analysis 100)

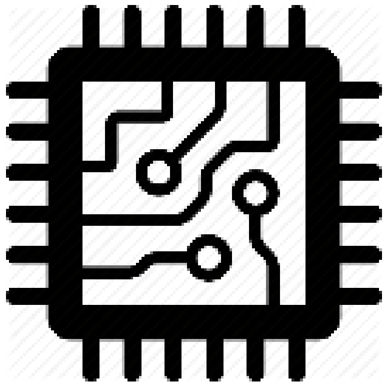


Physical attacks

Side-channel analysis

Fault injection attacks

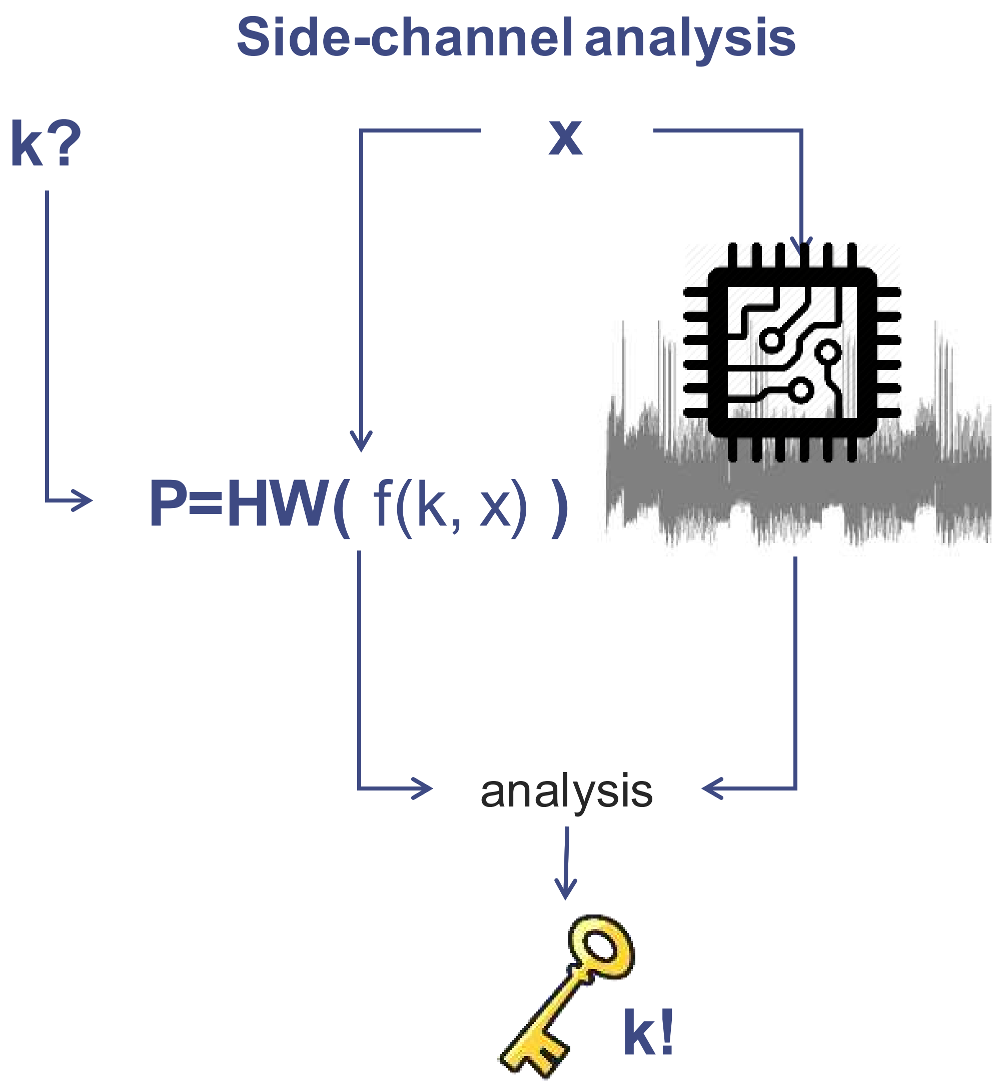


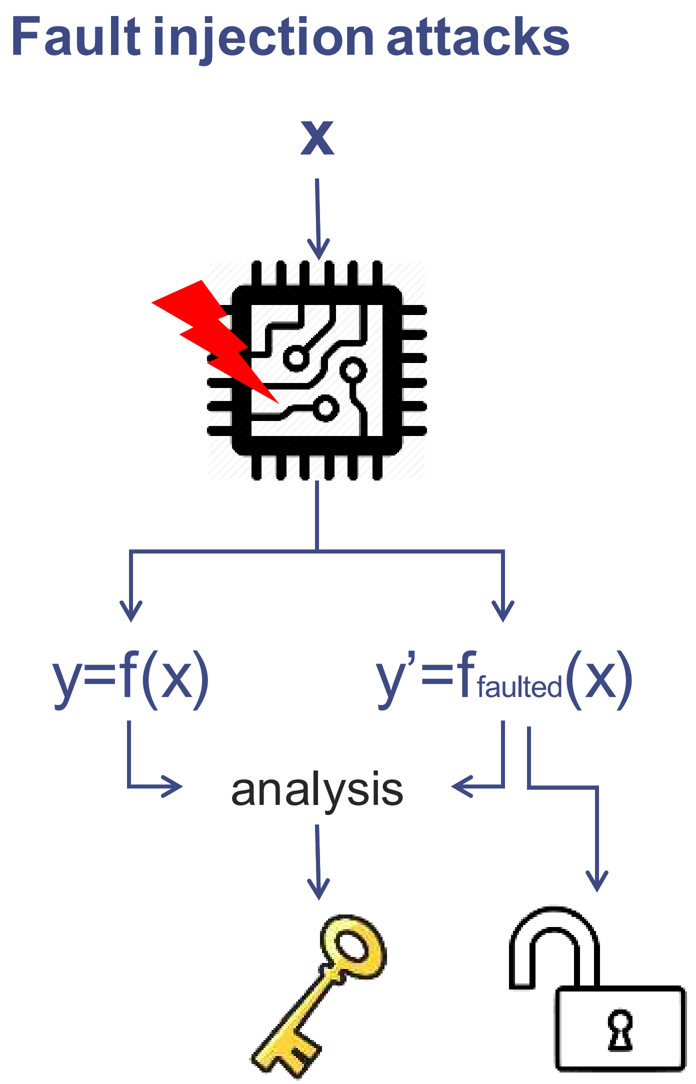


Highly effective against cryptographic implementations

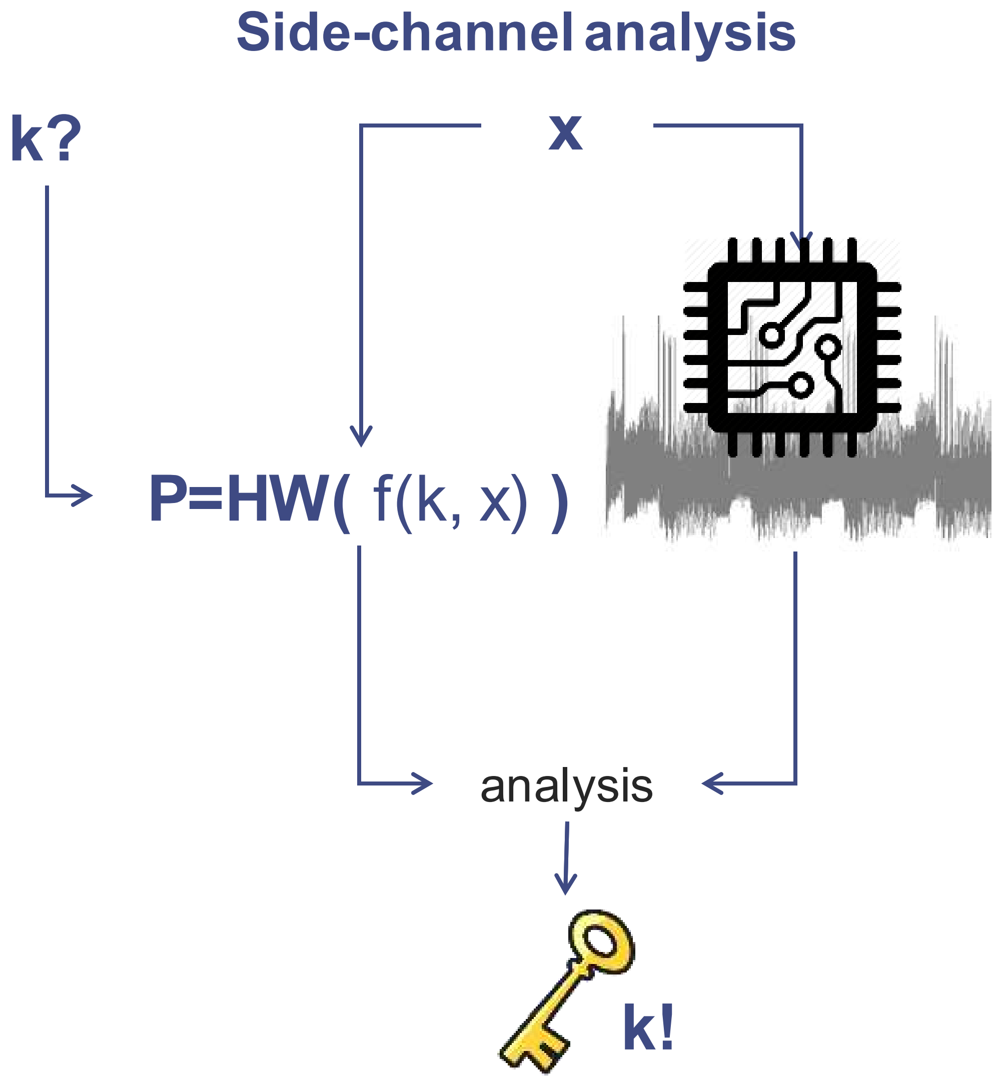
Can leverage software vulnerabilities [Cui & Rousley, 2017]

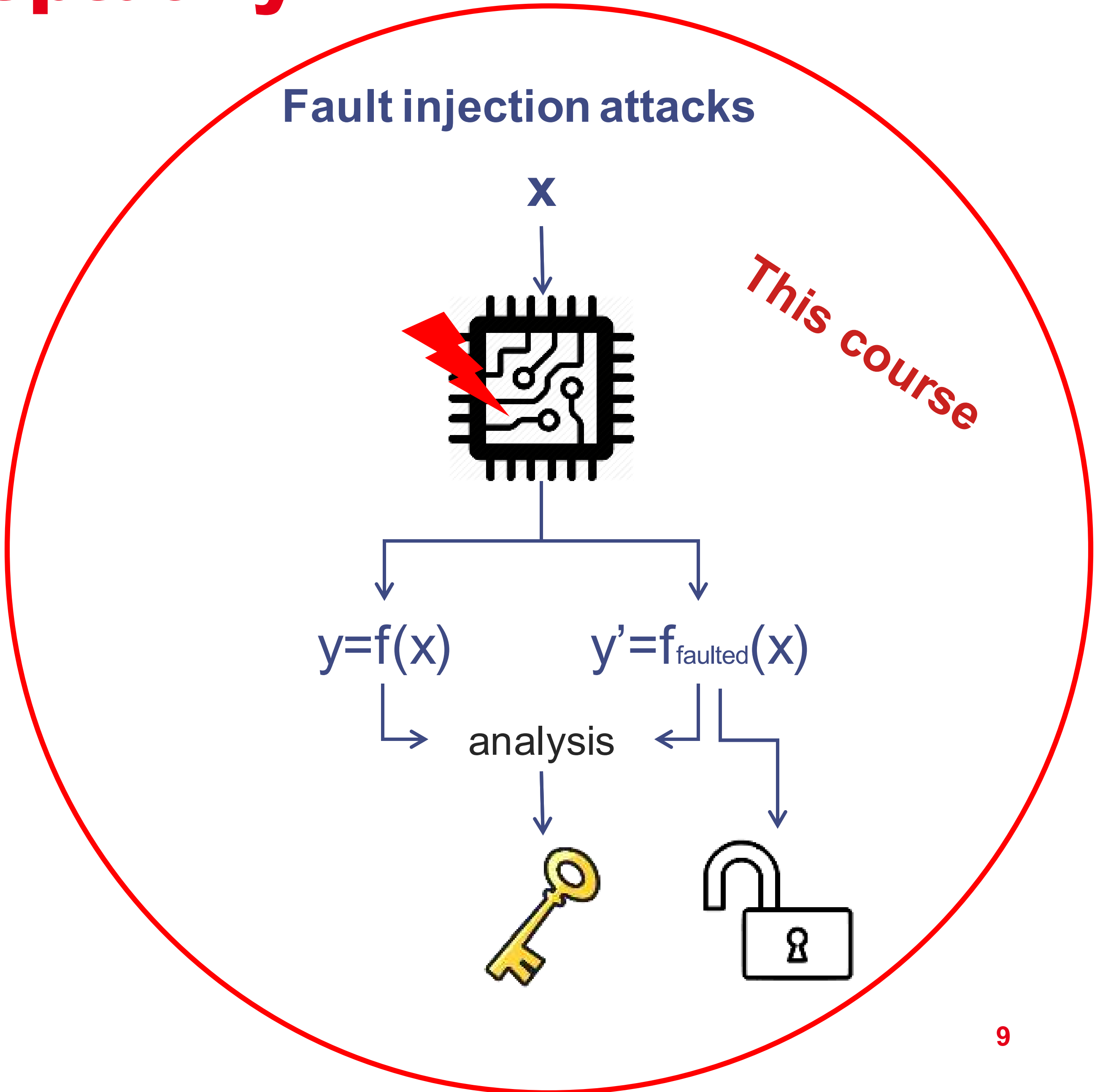
Physical attacks, conceptually





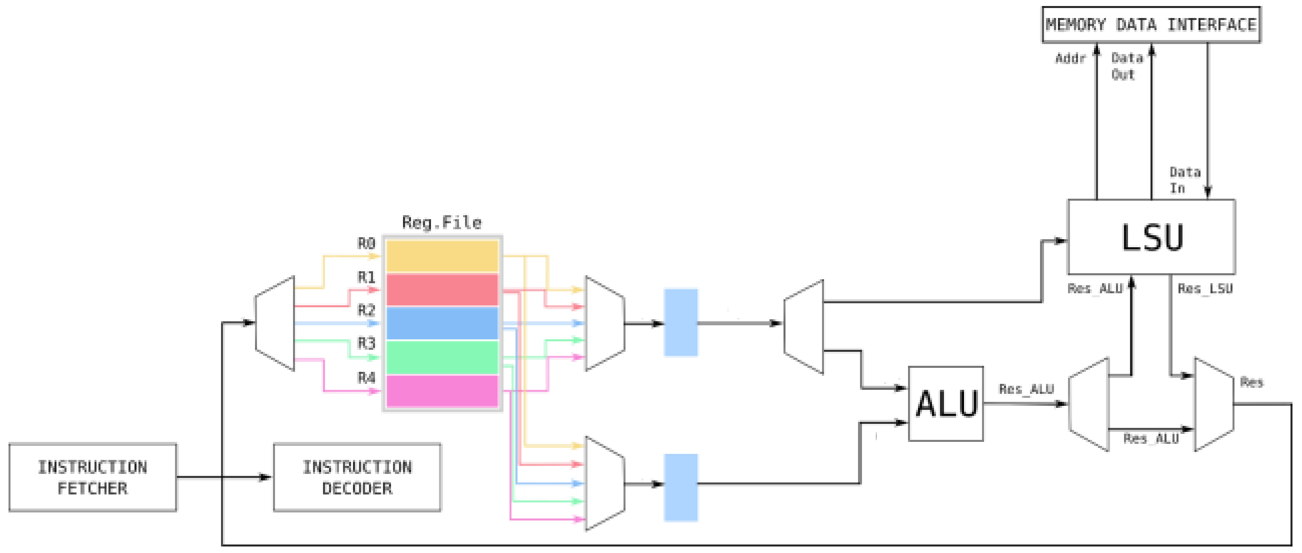
Physical attacks, conceptually





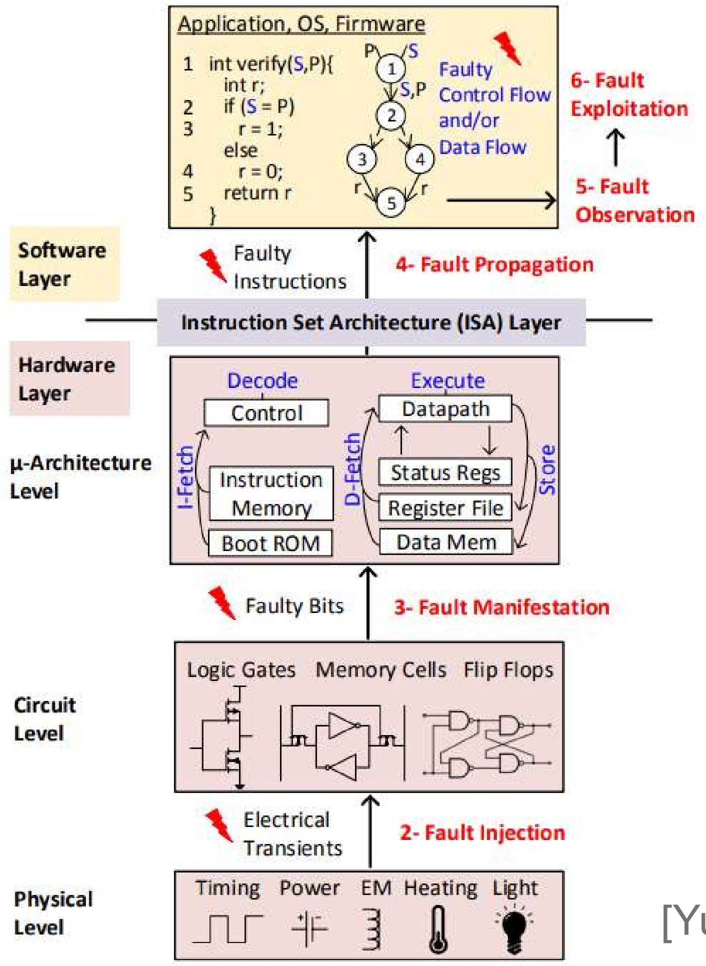
Physical attacks, behind the scenes

Side-channel analysis



[Casalino PhD, 2024]

Fault injection attacks



[Yuce & al. 2018]

Takeway

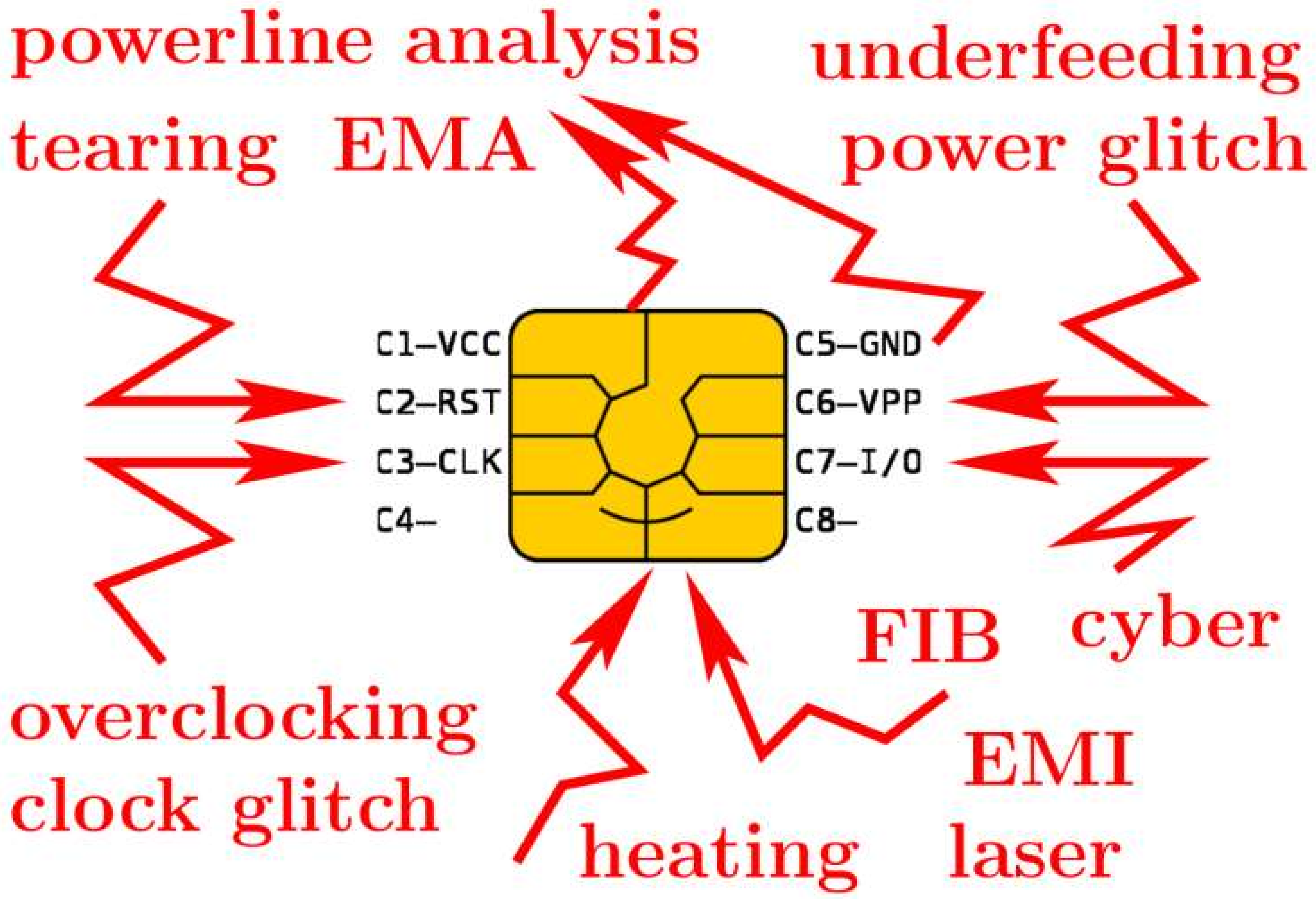
“ **Hardware security matters…**

(… and is challenging, but fun!)

Cybersecurity is not only a matter of maths and crypto, it also involves the physical nature of computing, via observation and perturbation.

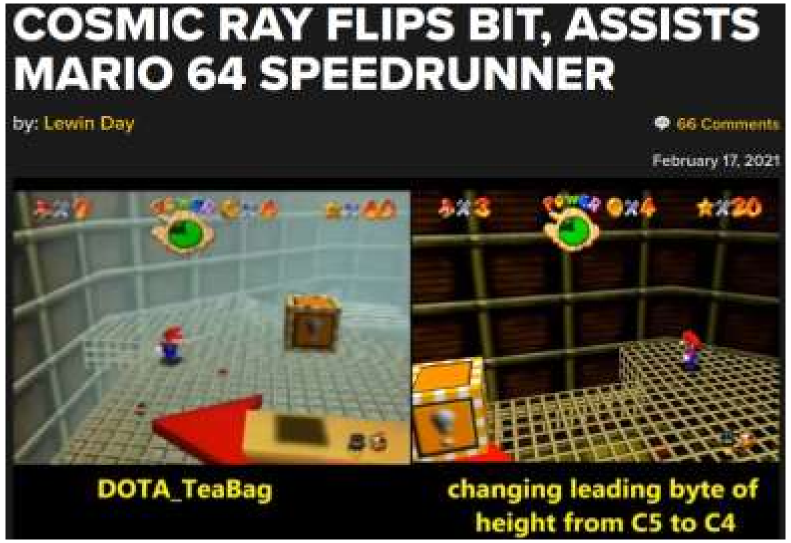
Fault Injection Attacks in the Cybersecurity Landscape

Hardware security in medieval times



Courtesy of Sylvain Guilley 2015, Télécom ParisTech - Secure-IC

Fault Injection: Safety, Security



COSMIC RAY FLIPS BIT, ASSISTS MARIO 64 SPEEDRUNNER

by: Lewin Day

66 Comments

February 17, 2021

Mario 64 SpeedRunner

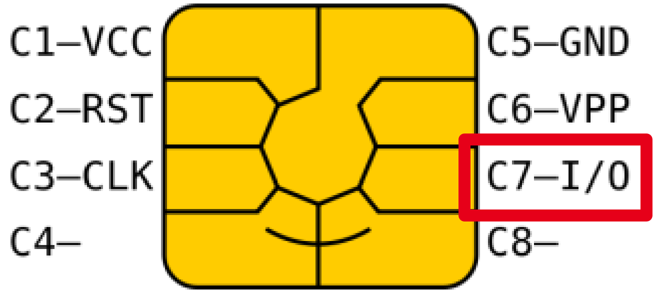
Xbox360 reset glitch hack
https://free60.org/Hacks/Reset_Glitch_Hack

	Safety	Security
fault injection	natural / non intentional	intentional
verification method	probabilistic	exhaustive

From the Secure Element to the IoT

Secure Element

- The HW and SW architecture is carried out by one main provider − and possibly a few sub-contractors
- Limited connectivity and communication capabilities
- Logical attacks are considered, but are not the main threat.
- **Considered secured**; security evaluated by expensive certification processes before market deployment
- **Impacts of a security breach: mostly limited** to the exploitation of the data stored in the component.



IoT device

- Integration of many HW and SW components, mostly issued by (untrusted) third parties.
- Lots of communication and sensing capabilities
- **Known to be unsecured**; lots of potential security vulnerabilities, certification is still an open topic and available schemes (e.g. CSPN) are not widely adopted.
- Impacts of a security breach:
  - Device level: usually low. On-device data have low value.
  - **Network/infrastructure level: high**. The device can be used as a stepping stone to attack other systems.
  - **Societal level: high**. Discredits the use of technology.

Fault injection Target

- Real life attack
  - Xbox360 reset glitch hack (2011) [1]
    - Allows unsigned code execution

Reset Glitch Hack (RGH) is a hardware modification which allows you to **run unsigned code, mods, game backups, and homebrew.** The hack relies on a **vulnerability in the hardware** found by GliGli that is triggered by **sending a reset pulse to the processor** at a specific moment, resulting in a power glitch that causes a bootloader hash check to return "valid" no matter what you have flashed in place of the stock bootloader. The timing of when and how long the pulse should be sent is dependent on the console and it may take some tweaking until it "glitches" and boots.
https://consolemods.org/wiki/Xbox_360:RGH

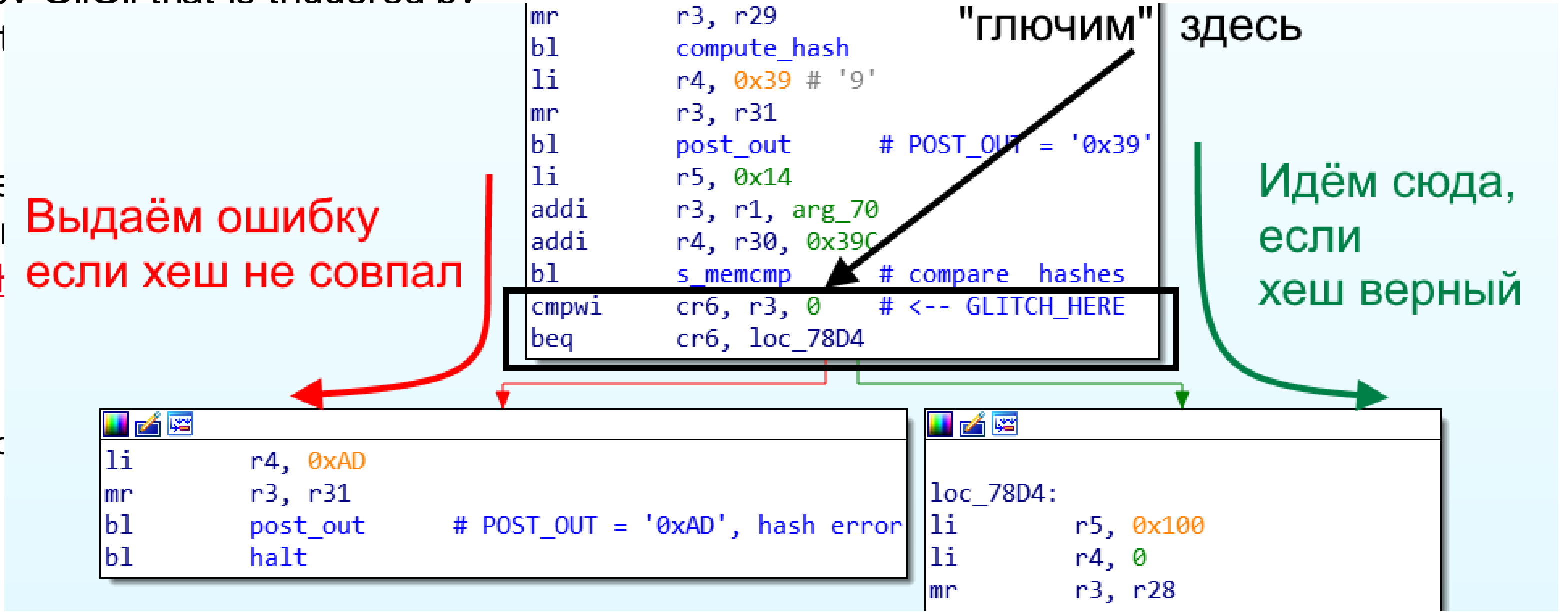
[1] https://free60project.github.io/wiki/Reset_Glitch_Hack

Fault injection Target

- Real life attack
  - Xbox360 reset glitch hack (2011) [1]
    - Allows unsigned code execution

Reset Glitch Hack (RGH) is a hardware modification which allows you to **run unsigned code, mods, game backups, and homebrew.** The hack relies on a **vulnerability in the hardware** found by GliGli that is triggered by **sending a reset pulse to the processor** at power glitch that causes a bootloader hash what you have flashed in place of the stock and how long the pulse should be sent is de may take some tweaking until it "glitches" ar
https://consolemods.org/wiki/Xbox_360:RGH

[1] https://free60project.github.io/wiki/Reset_Glitch_Hac



```
mr      r3, r29
bl      compute_hash
li      r4, 0x39 # '9'
mr      r3, r31
bl      post_out       # POST_OUT = '0x39'
li      r5, 0x14
addi    r3, r1, arg_70
addi    r4, r30, 0x39C
bl      s_memcmp       # compare  hashes
cmpwi   cr6, r3, 0     # <-- GLITCH_HERE
beq     cr6, loc_78D4
```

```
li      r4, 0xAD
mr      r3, r31
bl      post_out       # POST_OUT = '0xAD', hash error
bl      halt
```

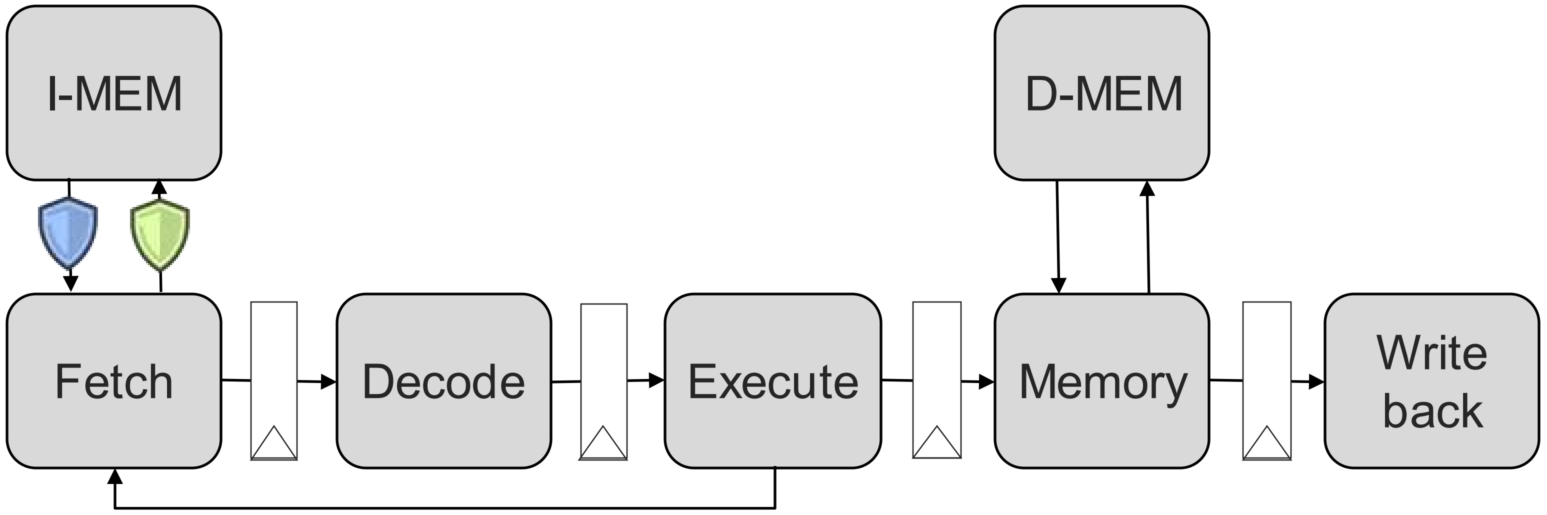
```
loc_78D4:
li      r5, 0x100
li      r4, 0
mr      r3, r28
```

Fault injection Target

- Real life attack
  - Xbox360 reset glitch hack (2011) [1]
    - Allows unsigned code execution
- Targets evolution
  - 8 bits AVR micro controller (2011) [2]
  - 32 bits dualcore ARM Cortex A9 (2019) [3]
  - BCM2837 32 bits quadcore ARM Cortex (2019) A53 [4]
  - Mobile devices [5]

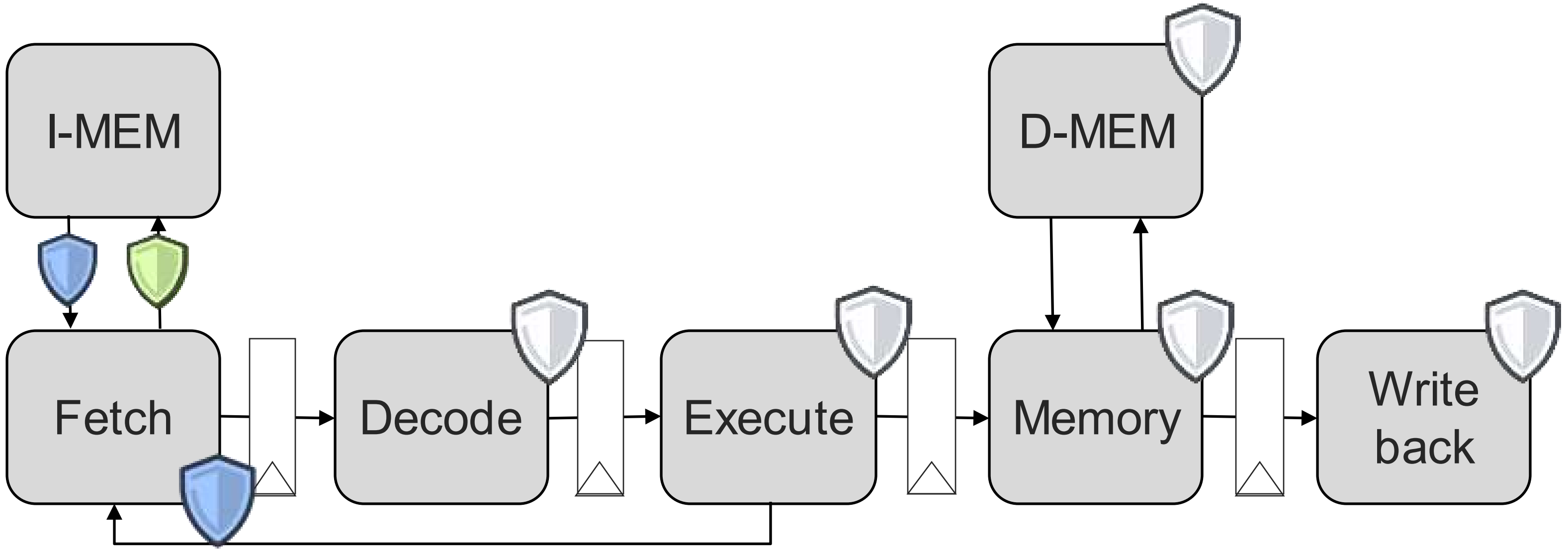
[1] https://free60project.github.io/wiki/Reset_Glitch_Hack/
[2] Balasch, J. et al, "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs.“, 2011 FDTC
[3] Proy, J. et al, "Studying EM Pulse Effects on Superscalar Microarchitectures at ISA Level.“, 2019 CoRR
[4] Trouchkine, et al. "Fault Injection Characterization on Modern CPUs.“, 2019 WISTP
[5] “Physical Fault Injection and Side-Channel Attacks on Mobile Devices : A Comprehensive Analysis”, Computers & Security (2021). https://doi.org/10.1016/j.cose.2021.102471

What to Protect?
State of the Art: Security Properties



What to Protect?
State of the Art: Security Properties

- Data integrity
- Code authenticity / integrity
- Control-flow integrity
  - Direct branches / calls
  - Indirect branches / calls
  - Branchless instructions sequences (a.k.a. *basic blocks*)
    - Execution of all the instructions (e.g. no skip)
    - In correct order

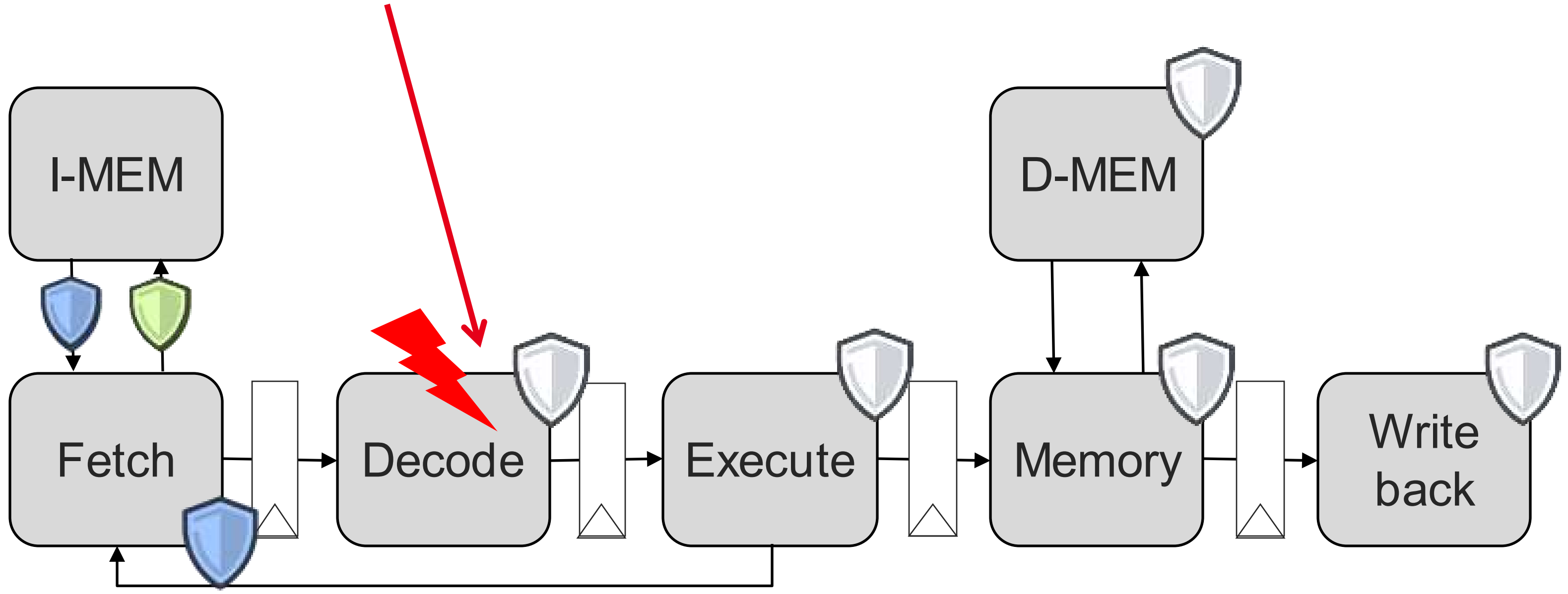


Problem: faults targeting control signals

A simple loop code:

```
loop:
    addi t0, t0, #-1
    ~~bne~~ t0, zero, loop
```

beq ← **Fault on instruction decode**

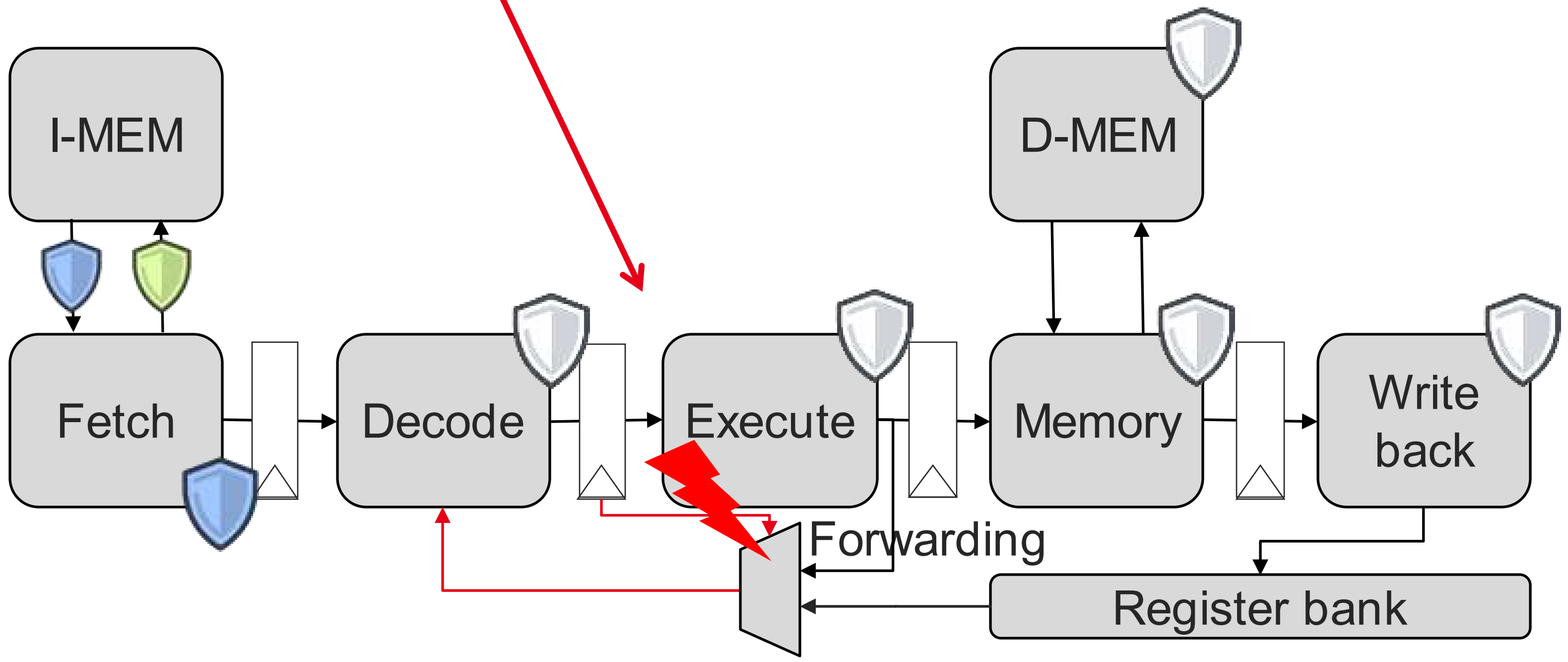


Problem: faults targeting control signals

A simple loop code:

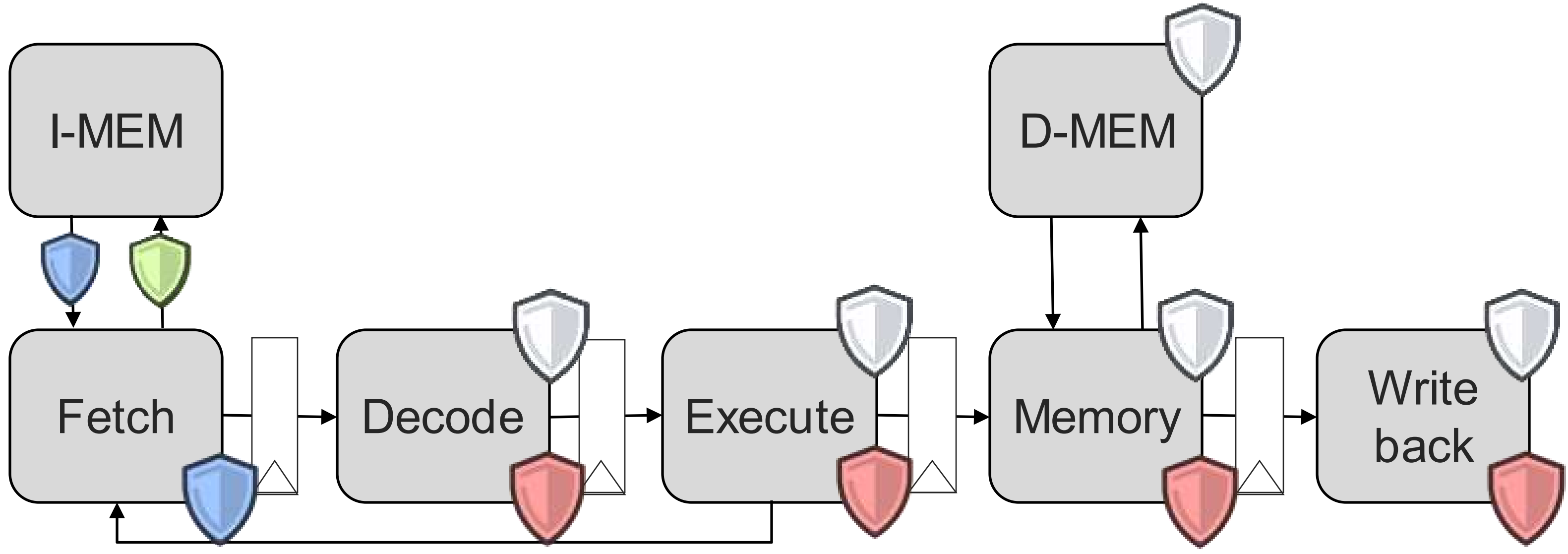
```
loop:
    addi t0[n], t0[n-1], #-1
    bne ~~t0[n]~~, zero, loop
```

t0[n-1] ← **Fault on forwarding**



What to Protect?
State of the Art: Security Properties

- Data integrity
- Code authenticity / integrity
- Control-flow integrity
  - Direct branches / calls
  - Indirect branches / calls
  - Branchless instructions sequences (a.k.a. *basic blocks*)
    - Execution of all the instructions (e.g. no skip)
    - In correct order
- **Control Signals Integrity**



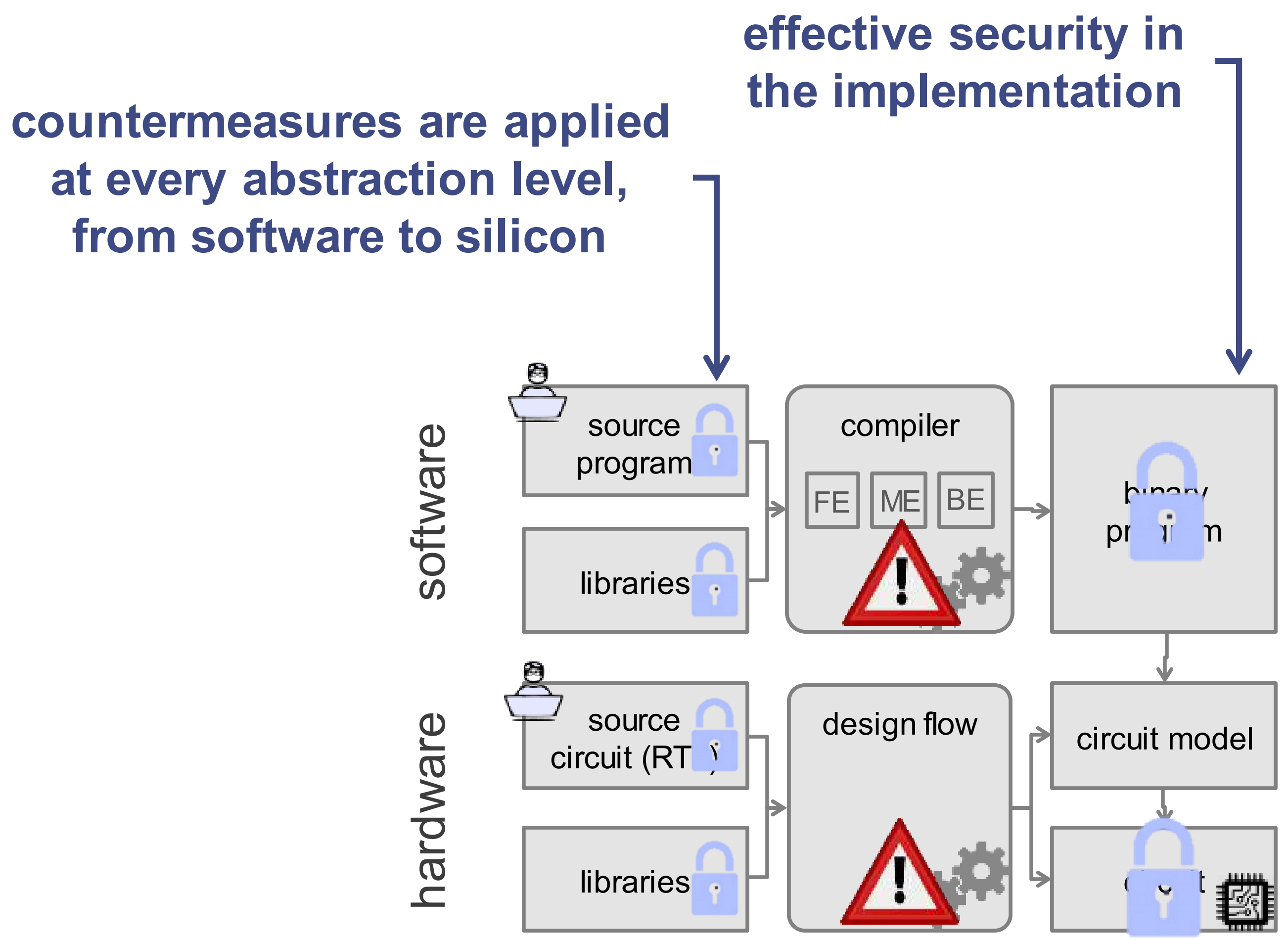
Takeway

“ Fault injection attacks increase the attack surface of embedded systems, and require protections down to the processor microarchitecture.

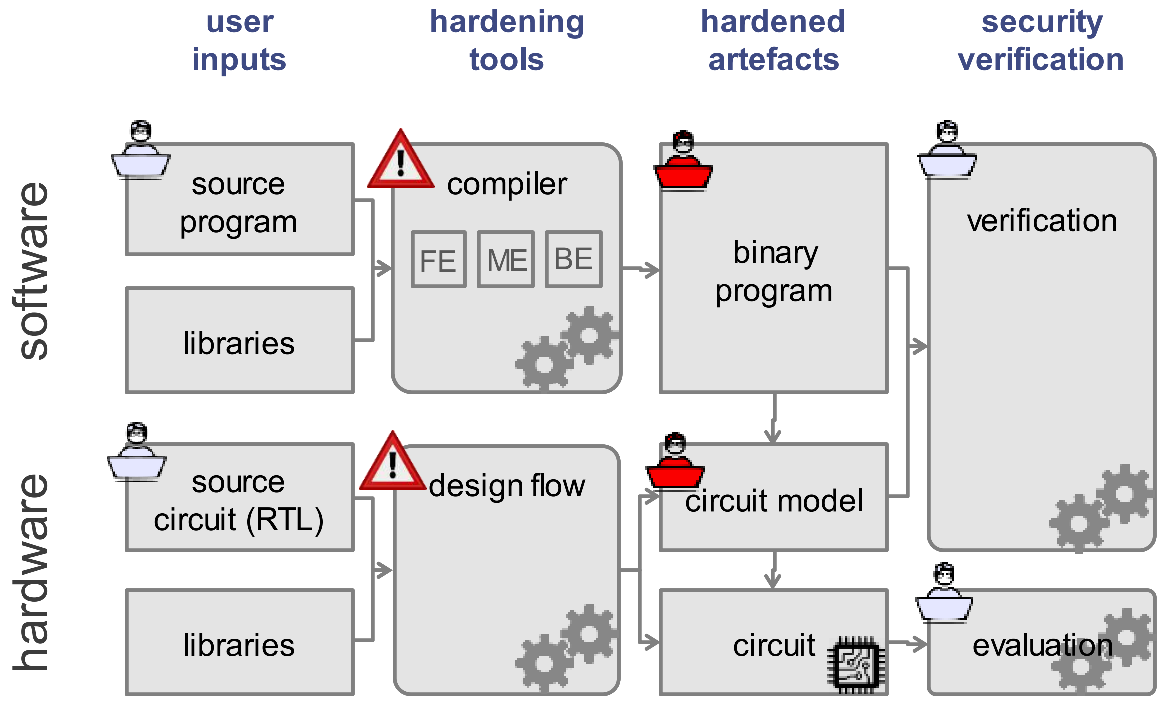
Workflow production for numerical systems

the role of the compiler

Towards the production of secure digital systems



Towards the production of secure digital systems



Common Criteria

Takeway

“ **The compiler sits between the developer and the final, concrete product. You must consider carefully it’s impact on countermeasures.**

(synthesis tools: same story!)

■ Compilers vs. Security

Compiler duties & objectives

Duties: assurance of functional equivalence between source code and machine code

- “functional” / “functionality” is usually not precisely defined
  - Side effects?
  - Determinism of time behaviour? (real time execution)
  - Lazy evaluation?
- No formal assurance
  - Except few works, such as CompCert
- Correctness by construction?
  - The source code written by the developper is not always valid

Objectives: optimise one or several performance criteria

- Execution time
- Resources: e.g. memory consumption
- Energy consumption, power consumption
- There is no complete criterion for optimality, and no convergence
  - Nature of the algorithm used
  - Relation to architecture / micro-architecture
  - Data

Compiler rights

Rights

- Reorganise contents of the target program, as long as program semantics is preserved
  - Machine instructions, basic blocs
- Select the best translation for a source code operation / instruction
- Remove parts of the program, as long as the program functionality is considered to be preserved (i.e. the computation does not participate in producing the program results)

Some classical optimisation passes:

- *dead code elimination*
- *global value numbering*
- common-subexpression elimination
- *strength reduction*
- *loop strength reduction, loop simplification, loop-invariant code motion*

LLVM’s Analysis and Transform Passes, the 2016/06/30

- 40 analysis passes
- 56 transformation/optimisation passes
- 10 utilitary passes
- … backends, etc.

Compiler rights

Rights

- Reorganise contents of the target program, as long as program semantics is preserved
  - Machine instructions, basic blocs
- Select the best translation for a source code operation / instruction
- Remove parts of the program, as long as the program functionality is considered to be preserved (i.e. the computation does not participate in producing the program results)

Some classical optimisation passes:

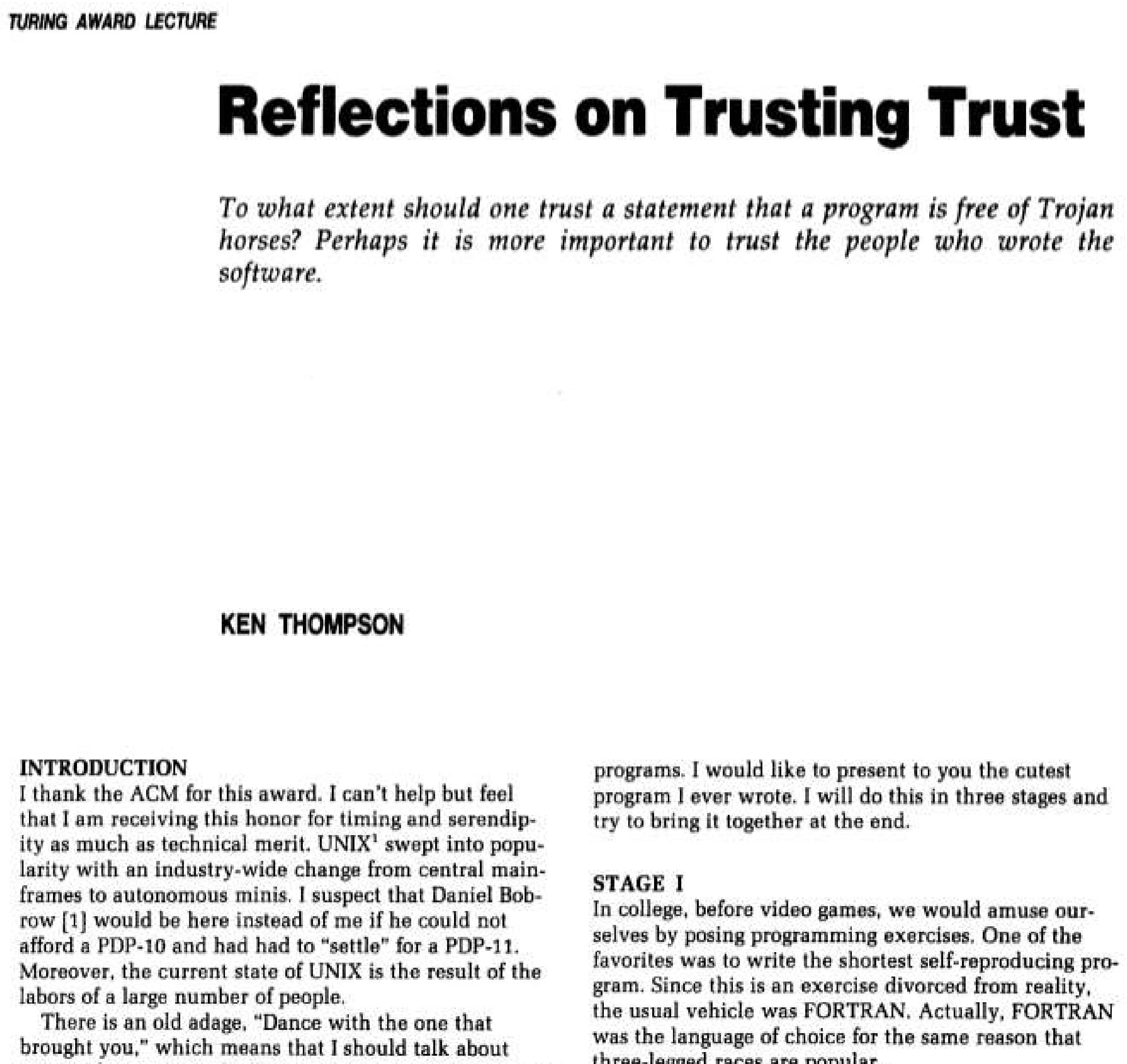
- *dead code elimination*
- *global value numbering*
- common-subexpression elimination
- *strength reduction*
- *loop strength reduction, loop simplification, loop-invariant code motion*

Will break your security

LLVM's Analysis and Transform Passes, the 2016/06/30

- 40 analysis passes
- 56 transformation/optimisation passes
- 10 utilitary passes
- … backends, etc.

A must read



TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION
I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

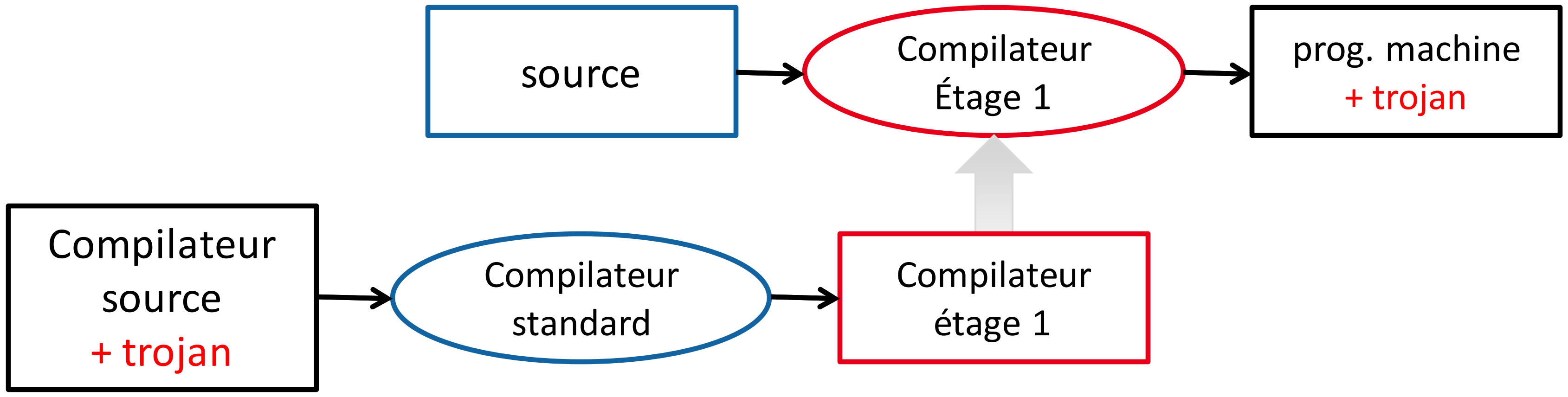
There is an old adage, "Dance with the one that brought you," which means that I should talk about programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I
In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

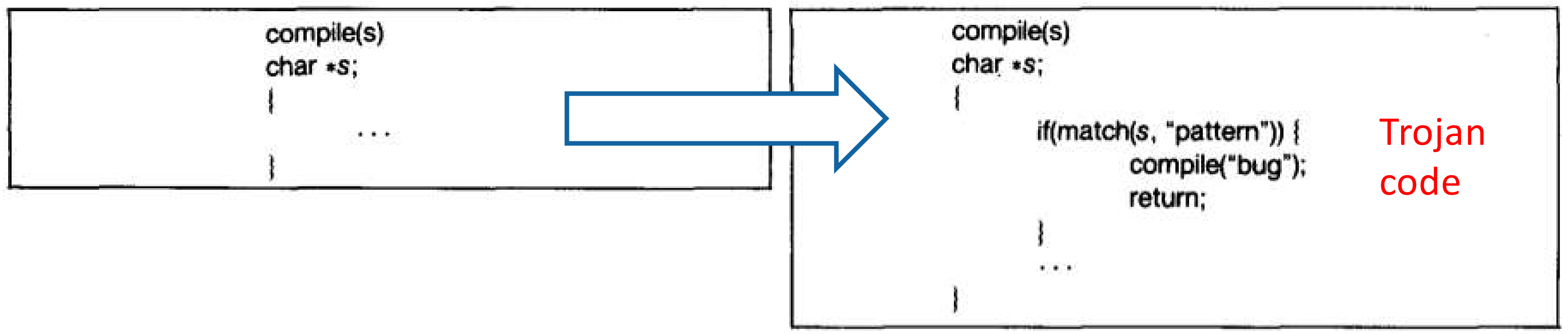
Ken Thompson
Communications of the ACM
August 1984,
vol 28 number 8

Kenneth Lane Thompson is an American pioneer of computer science. Thompson worked at Bell Labs for most of his career where he designed and implemented the original **Unix** operating system. He also invented the **B programming language**, the direct predecessor to the C language, and was one of the creators and early developers of the Plan 9 operating system. Other notable contributions included his work on regular expressions and early computer text editors QED and **ed**, the definition of the UTF-8 encoding, and his work on computer chess that included the creation of endgame tablebases and the chess machine Belle. Since 2006, Thompson has worked at Google, where he co-developed the **Go** language.

Reflections on trusting trust. Insertion silencieuse de trojans



Compilateur étage 1:

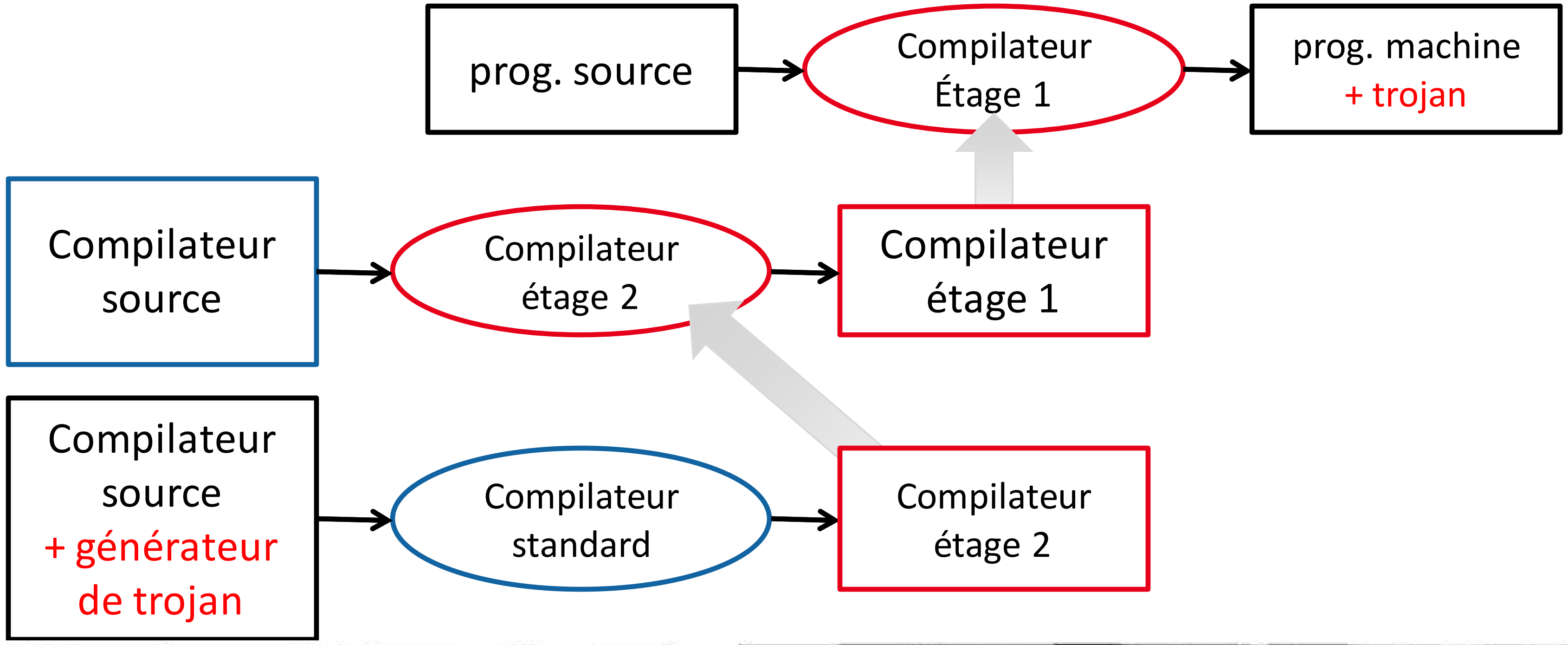


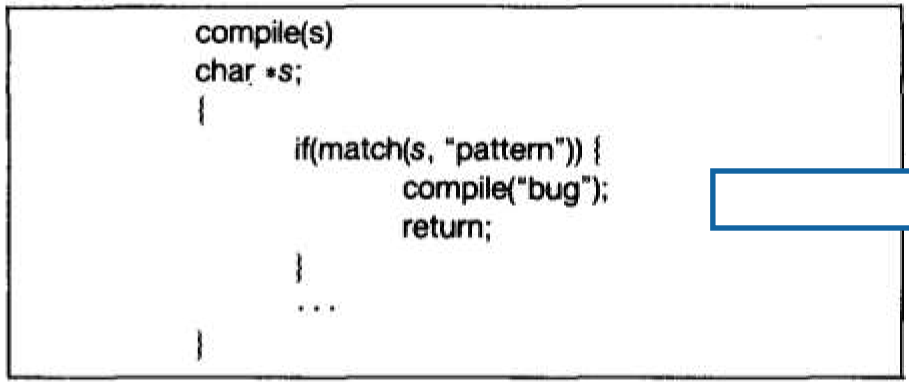
```
compile(s)
char *s;
{
        ...
}
```

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

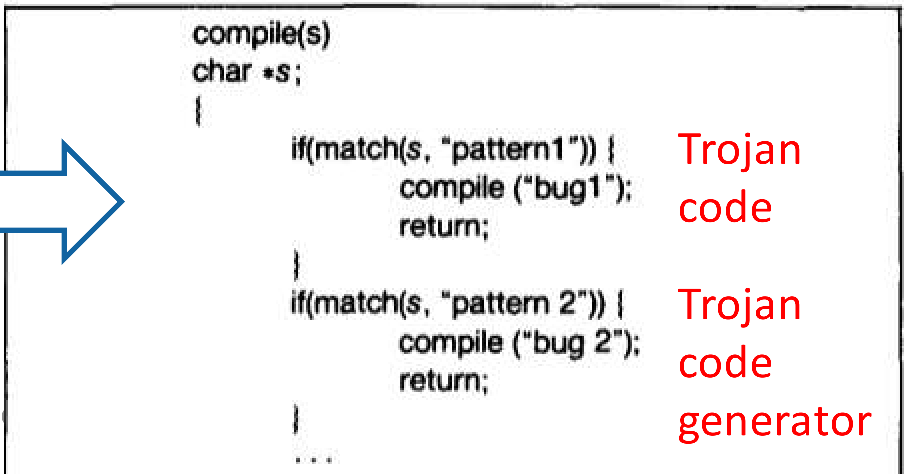
Trojan code

Reflections on trusting trust. Insertion silencieuse de trojans





```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        . . .
}
```

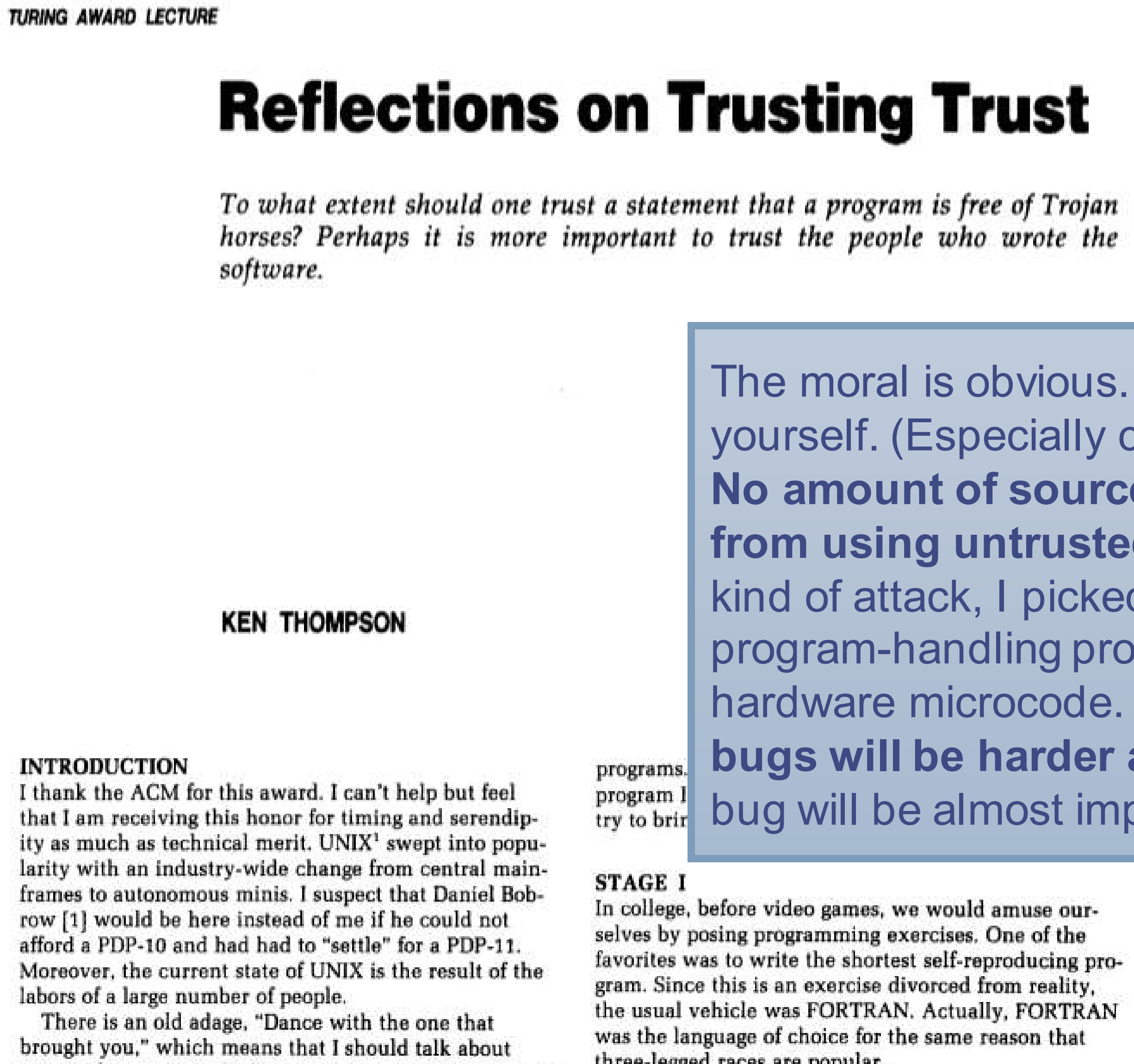


```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        . . .
```

Trojan code

Trojan code generator

A must read



TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION
I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about

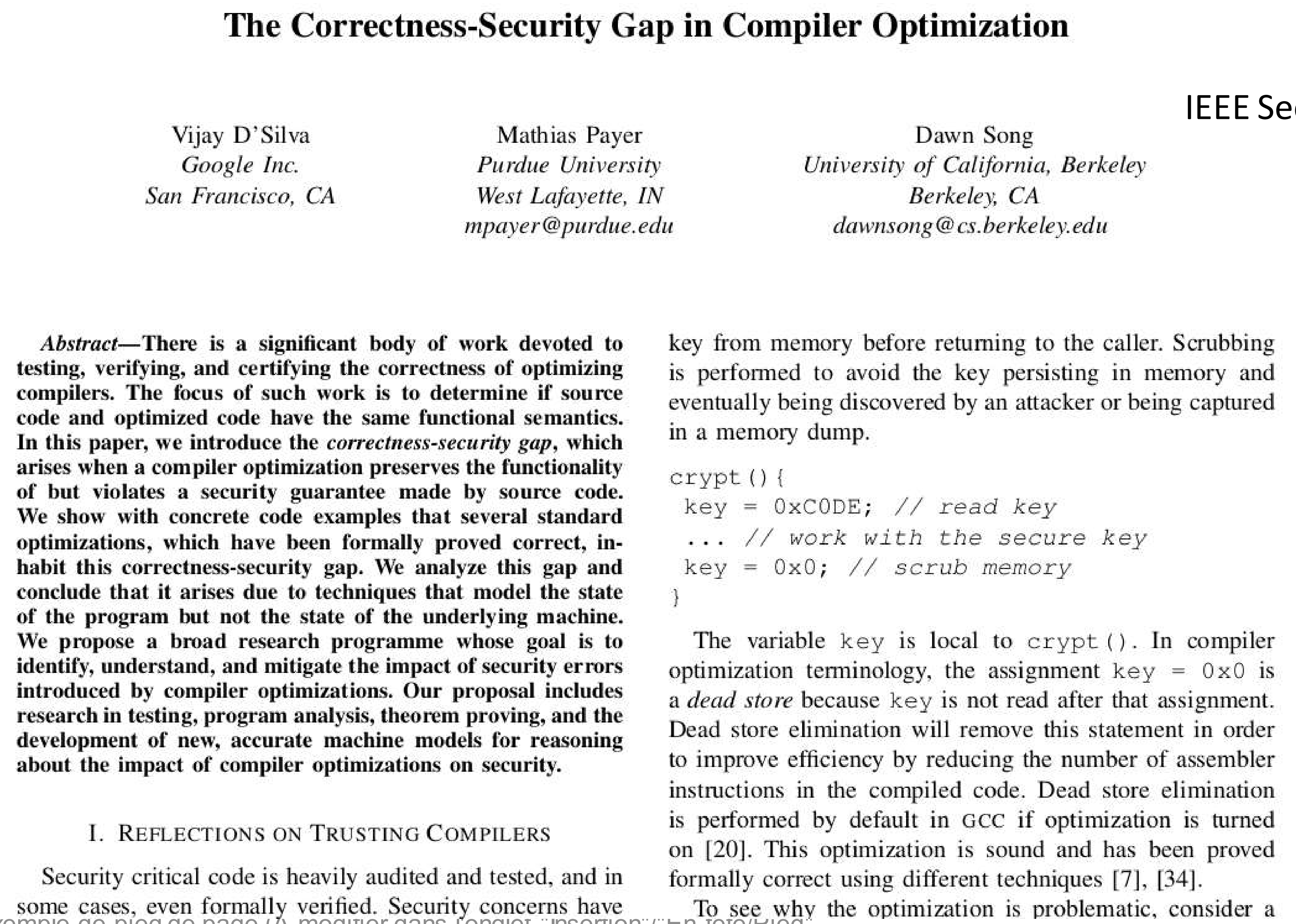
programs.
program I
try to brir

STAGE I
In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

Communications of the ACM
August 1984,
vol 28 number 8

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) **No amount of source-level verification or scrutiny will protect you from using untrusted code**. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. **As the level of program gets lower, these bugs will be harder and harder to detect.** A well-installed microcode bug will be almost impossible to detect.

Another must read



The Correctness-Security Gap in Compiler Optimization

Vijay D'Silva
Google Inc.
San Francisco, CA

Mathias Payer
Purdue University
West Lafayette, IN
mpayer@purdue.edu

Dawn Song
University of California, Berkeley
Berkeley, CA
dawnsong@cs.berkeley.edu

IEEE Security & Privacy Workshops 2015

***Abstract*—There is a significant body of work devoted to testing, verifying, and certifying the correctness of optimizing compilers. The focus of such work is to determine if source code and optimized code have the same functional semantics. In this paper, we introduce the *correctness-security gap*, which arises when a compiler optimization preserves the functionality of but violates a security guarantee made by source code. We show with concrete code examples that several standard optimizations, which have been formally proved correct, in-habit this correctness-security gap. We analyze this gap and conclude that it arises due to techniques that model the state of the program but not the state of the underlying machine. We propose a broad research programme whose goal is to identify, understand, and mitigate the impact of security errors introduced by compiler optimizations. Our proposal includes research in testing, program analysis, theorem proving, and the development of new, accurate machine models for reasoning about the impact of compiler optimizations on security.**

I. Reflections on Trusting Compilers

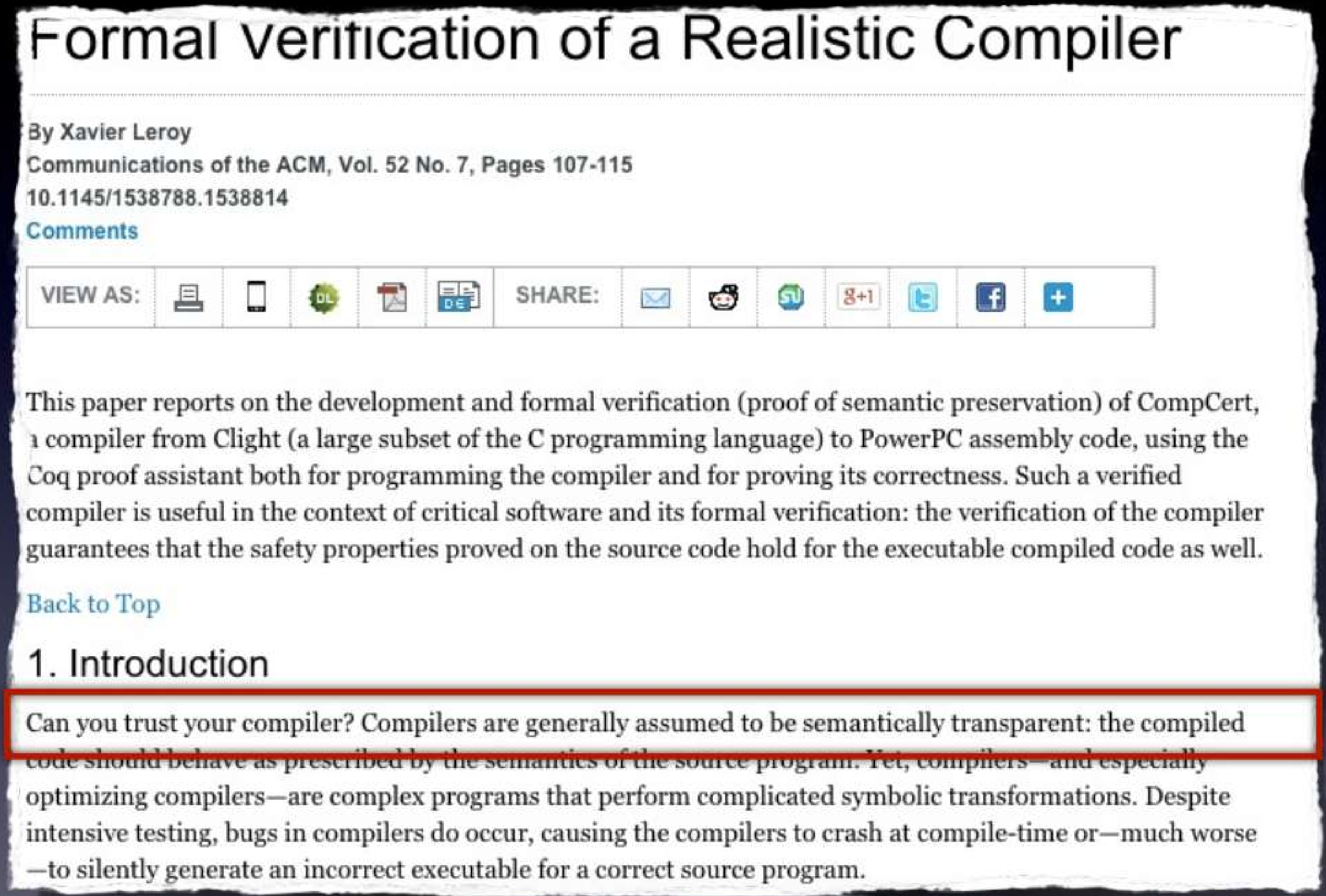
Security critical code is heavily audited and tested, and in some cases, even formally verified. Security concerns have

key from memory before returning to the caller. Scrubbing is performed to avoid the key persisting in memory and eventually being discovered by an attacker or being captured in a memory dump.

```
crypt(){
 key = 0xC0DE; // read key
 ... // work with the secure key
 key = 0x0; // scrub memory
}
```

The variable `key` is local to `crypt()`. In compiler optimization terminology, the assignment `key = 0x0` is a *dead store* because `key` is not read after that assignment. Dead store elimination will remove this statement in order to improve efficiency by reducing the number of assembler instructions in the compiled code. Dead store elimination is performed by default in GCC if optimization is turned on [20]. This optimization is sound and has been proved formally correct using different techniques [7], [34].

To see why the optimization is problematic, consider a



Formal Verification of a Realistic Compiler

By Xavier Leroy
Communications of the ACM, Vol. 52 No. 7, Pages 107-115
10.1145/1538788.1538814
Comments

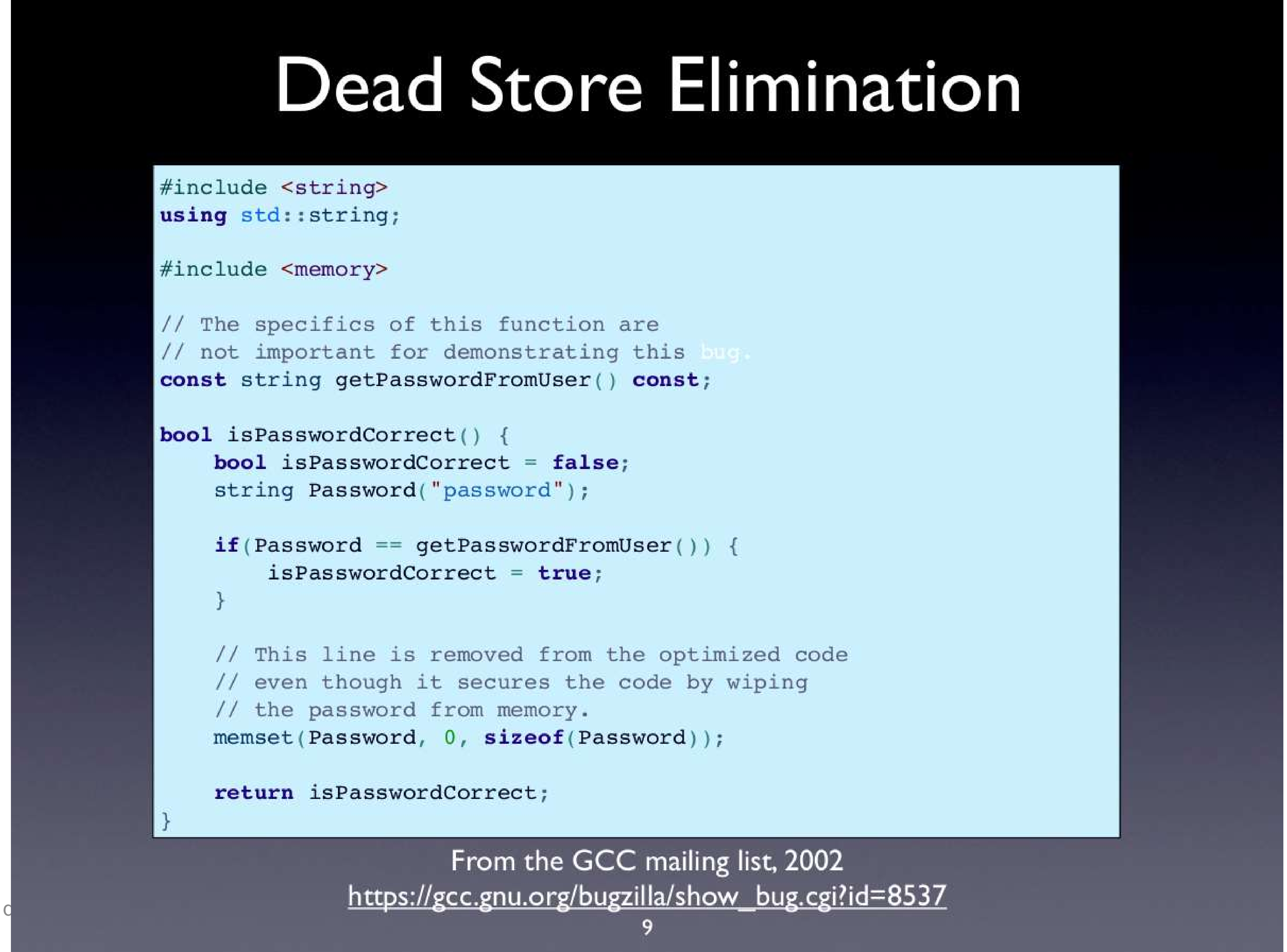
VIEW AS: SHARE:

This paper reports on the development and formal verification (proof of semantic preservation) of CompCert, a compiler from Clight (a large subset of the C programming language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Such a verified compiler is useful in the context of critical software and its formal verification: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Back to Top

1. Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compilers to crash at compile-time or—much worse—to silently generate an incorrect executable for a correct source program.

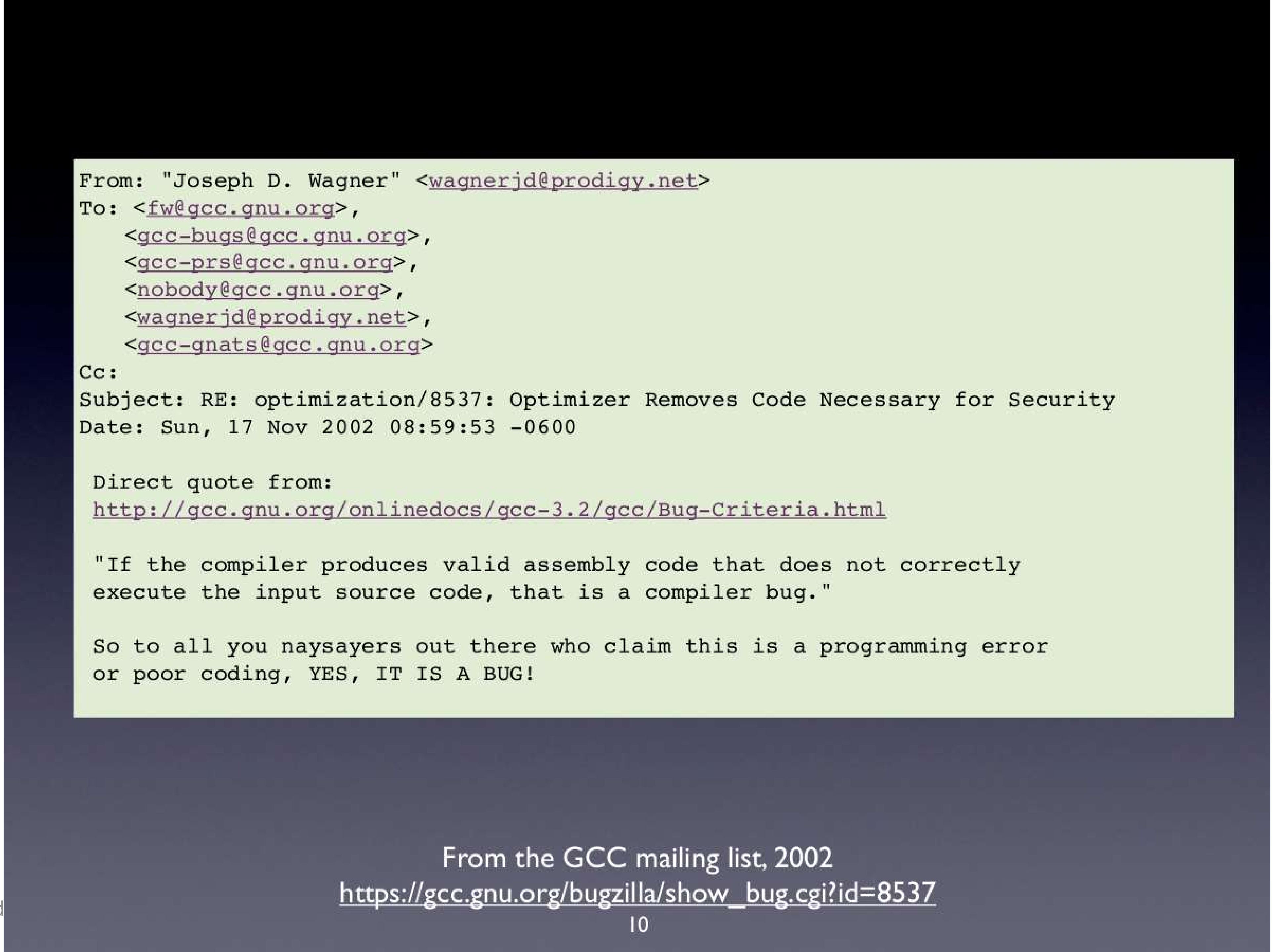


Dead Store Elimination

```cpp
#include <string>
using std::string;

#include <memory>

// The specifics of this function are
// not important for demonstrating this bug.
const string getPasswordFromUser() const;

bool isPasswordCorrect() {
    bool isPasswordCorrect = false;
    string Password("password");

    if(Password == getPasswordFromUser()) {
        isPasswordCorrect = true;
    }

    // This line is removed from the optimized code
    // even though it secures the code by wiping
    // the password from memory.
    memset(Password, 0, sizeof(Password));

    return isPasswordCorrect;
}
```

From the GCC mailing list, 2002

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537



```
From: "Joseph D. Wagner" <wagnerjd@prodigy.net>
To: <fw@gcc.gnu.org>,
   <gcc-bugs@gcc.gnu.org>,
   <gcc-prs@gcc.gnu.org>,
   <nobody@gcc.gnu.org>,
   <wagnerjd@prodigy.net>,
   <gcc-gnats@gcc.gnu.org>
Cc:
Subject: RE: optimization/8537: Optimizer Removes Code Necessary for Security
Date: Sun, 17 Nov 2002 08:59:53 -0600

 Direct quote from:
 http://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Bug-Criteria.html

 "If the compiler produces valid assembly code that does not correctly
 execute the input source code, that is a compiler bug."

 So to all you naysayers out there who claim this is a programming error
 or poor coding, YES, IT IS A BUG!
```

From the GCC mailing list, 2002
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537

10

Compiler interference

- **Optimization effects**
  - Preserve code functionality,
  - But do not preserve non-functional properties (e.g., execution time, etc.)
- **Several abstraction levels**

Compiler interference

- **Optimization effects**
  - Preserve code functionality,
  - But do not preserve non-functional properties (e.g., execution time, etc.)
- **Several abstraction levels**

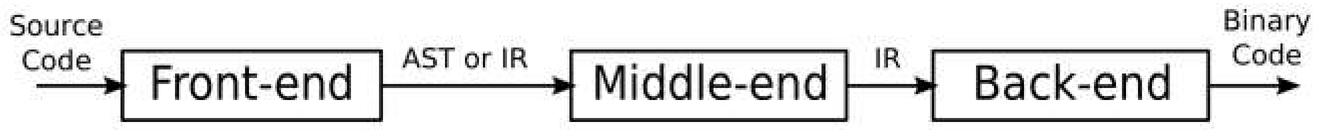
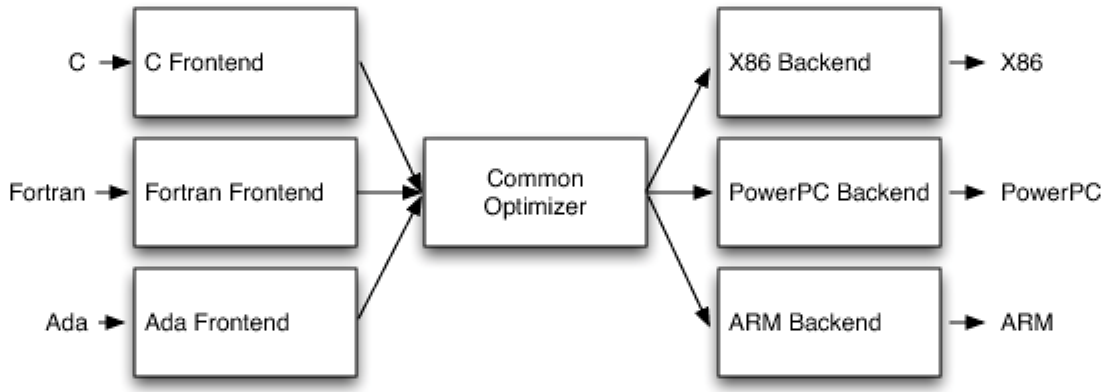


FIGURE 3.1: Components of a three-phase compiler.



[1] S. Tuan Vu, « Optimizing Property-Preserving Compilation », Doctoral thesis, Sorbonne University, 2021.

Compiler abstraction levels

- **Optimization effects**
  - Preserve code functionality,
  - But do not preserve non-functional properties (e.g., execution time, etc.)
- **Several abstraction levels**

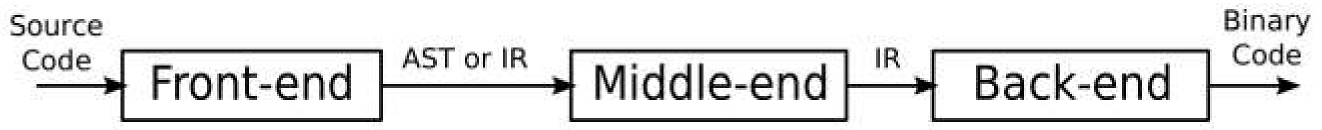


FIGURE 3.1: Components of a three-phase compiler.

IR → SelectionDAG → MachineDAG → MachineInstr → MCInst

[1] S. Tuan Vu, « Optimizing Property-Preserving Compilation », Doctoral thesis, Sorbonne University, 2021.
[2] https://releases.llvm.org/14.0.0/docs/WritingAnLLVMBackend.html

Compiler abstraction levels

- **Optimization effects**
  - Preserve code functionality,
  - But do not preserve non-functional properties (e.g., execution time, etc.)
- **Several abstraction levels**
- **Structural issues**
  - Some representations miss some information useful for security (but not needed for functionality).
    - E.g. SSA does not describe instruction ordering.

Takeway

“ A (standard) compiler only cares about functional properties,
it ~~may~~ will break security features.

Do not trust the compiler!

Compilers and security

Illustrated with simple examples

Insertion of dummy instructions

- Inserting a static procedure for desynchronisation

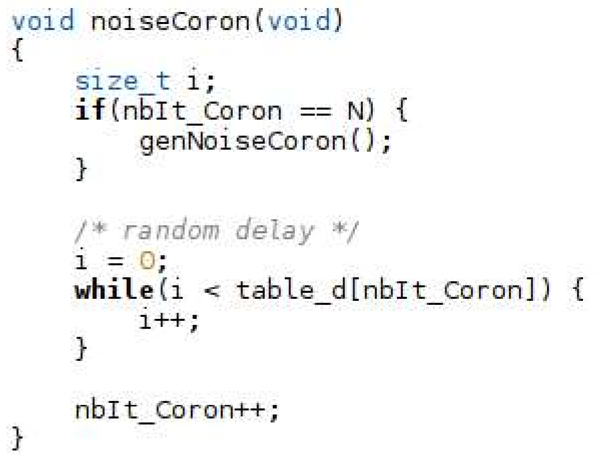
```
/* subBytes
 * Table Lookup
 */
void subBytes_f(void)
{
    int i;

    for(i = 0; i<16; i+=4)
    {
        CORON();
        state[i+0] = sbox[ state[i+0] ];
        state[i+1] = sbox[ state[i+1] ];
        state[i+2] = sbox[ state[i+2] ];
        state[i+3] = sbox[ state[i+3] ];
    }
}
```

```
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
    }

    nbIt_Coron++;
}
```

- Also possible (even better) with a timer and an interrupt handler

Coron, J. S., & Kizhvatov, I. An efficient method for random delay generation in embedded software. In Cryptographic Hardware and Embedded Systems-CHES 2009 (pp. 156-170). Springer (2009).
Coron, J.S., Kizhvatov, I. Analysis and improvement of the random delay countermeasure of CHES 2009. In: CHES. pp. 95–109. Springer (2010).

Insertion of dummy instructions



```c
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
    }

    nbIt_Coron++;
}
```

Compiled with -Os:

```
Dump of assembler code for function noiseCoron:
  0x0000859c <+0>:     push    {r4, lr}
  0x000085a0 <+4>:     ldr     r4, [pc, #28] ; <noiseCoron+40>
  0x000085a4 <+8>:     ldr     r3, [r4]    ; r3 ← nbIt_coron
  0x000085a8 <+12>:    cmp     r3, #160  ; nbIt_coron ?= N
  0x000085ac <+16>:    bne     0x85b4 <noiseCoron+24>
  0x000085b0 <+20>:    bl      0x8524 <genNoiseCoron>
  0x000085b4 <+24>:    ldr     r3, [r4]
  0x000085b8 <+28>:    add     r3, r3, #1 ; nbIt_coron++
  0x000085bc <+32>:    str     r3, [r4]
  0x000085c0 <+36>:    pop     {r4, pc}
  0x000085c4 <+40>:    andeq   r0, r1, r0, lsr r8
End of assembler dump.
```

Insertion of dummy instructions

```c
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
    }

    nbIt_Coron++;
}
```

Compiled with -Os:

```
Dump of assembler code for function noiseCoron:
   0x0000859c <+0>:      push    {r4, lr}
   0x000085a0 <+4>:      ldr     r4, [pc, #28] ; <noiseCoron+40>
   0x000085a4 <+8>:      ldr     r3, [r4]   ; r3 ← nbIt_coron
   0x000085a8 <+12>:     cmp     r3, #160  ; nbIt_coron ?= N
   0x000085ac <+16>:     bne     0x85b4 <noiseCoron+24>
   0x000085b0 <+20>:     bl      0x8524 <genNoiseCoron>
   0x000085b4 <+24>:     ldr     r3, [r4]
   0x000085b8 <+28>:     add     r3, r3, #1 ; nbIt_coron++
   0x000085bc <+32>:     str     r3, [r4]
   0x000085c0 <+36>:     pop     {r4, pc}
   0x000085c4 <+40>:     andeq   r0, r1, r0, lsr r8
End of assembler dump.
```

???

Insertion of dummy instructions

```c
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
    }

    nbIt_Coron++;
}
```

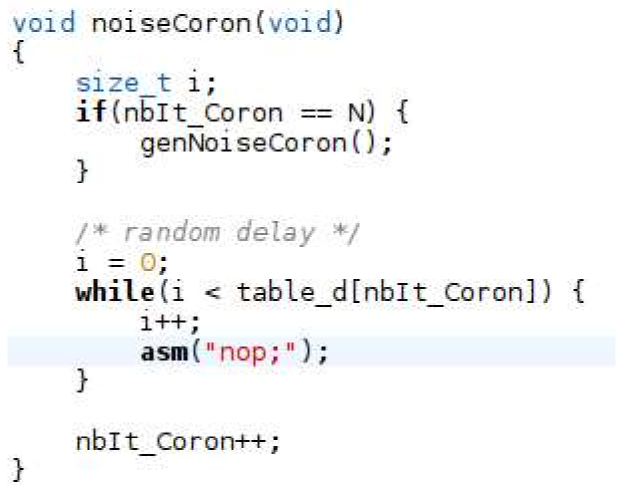
???

Compiled with -Os:

```
Dump of assembler code for function noiseCoron:
   0x0000859c <+0>:     push    {r4, lr}
   0x000085a0 <+4>:     ldr     r4, [pc, #28] ; <noiseCoron+40>
   0x000085a4 <+8>:     ldr     r3, [r4]     ; r3 ← nbIt_coron
   0x000085a8 <+12>:    cmp     r3, #160  ; nbIt_coron ?= N
   0x000085ac <+16>:    bne     0x85b4 <noiseCoron+24>
   0x000085b0 <+20>:    bl      0x8524 <genNoiseCoron>
   0x000085b4 <+24>:    ldr     r3, [r4]
   0x000085b8 <+28>:    add     r3, r3, #1 ; nbIt_coron++
   0x000085bc <+32>:    str     r3, [r4]
   0x000085c0 <+36>:    pop     {r4, pc}
   0x000085c4 <+40>:    andeq   r0, r1, r0, lsr r8
End of assembler dump.
```

???

Insertion of dummy instructions



```
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
        asm("nop;");
    }

    nbIt_Coron++;
}
```

Compiled with -Os:

```
Dump of assembler code for function noiseCoron:
   0x0000859c <+0>:   push   {r4, lr}
   0x000085a0 <+4>:   ldr    r4, [pc, #60]     ; <noiseCoron+72>
   0x000085a4 <+8>:   ldr    r3, [r4]
   0x000085a8 <+12>:  cmp    r3, #160     ; nbIt_coron ?= N
   0x000085ac <+16>:  bne    0x85b4 <noiseCoron+24>
   0x000085b0 <+20>:  bl     0x8524 <genNoiseCoron>
   0x000085b4 <+24>:  ldr    r3, [pc, #44]     ; <noiseCoron+76>
   0x000085b8 <+28>:  ldr    r2, [r4]
   0x000085bc <+32>:  ldr    r1, [r3, r2, lsl #2]
   0x000085c0 <+36>:  mov    r3, #0       ; i ← 0
   0x000085c4 <+40>:  cmp    r3, r1       ; i ?= nbIt_Coron
   0x000085c8 <+44>:  beq    0x85d8 <noiseCoron+60>
   0x000085cc <+48>:  add    r3, r3, #1   ; i ← i+1
   0x000085d0 <+52>:         nop
   0x000085d4 <+56>:  b      0x85c4 <noiseCoron+40>
   0x000085d8 <+60>:  add    r2, r2, #1   ; nbIt_Coron++
   0x000085dc <+64>:  str    r2, [r4]
   0x000085e0 <+68>:  pop    {r4, pc}
   0x000085e4 <+72>:  andeq  r0, r1, r4, asr r8
   0x000085e8 <+76>:  andeq  r0, r1, r12, asr r8
End of assembler dump.
```

Insertion of dummy instructions

```c
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
        asm("");
    }

    nbIt_Coron++;
}
```

Compiled with -Os:

```
Dump of assembler code for function noiseCoron:
   0x0000859c <+0>:  push     {r4, lr}
   0x000085a0 <+4>:  ldr      r4, [pc, #56] ; <noiseCoron+68>
   0x000085a4 <+8>:  ldr      r3, [r4]
   0x000085a8 <+12>: cmp      r3, #160    ; 0xa0
   0x000085ac <+16>: bne      0x85b4 <noiseCoron+24>
   0x000085b0 <+20>: bl       0x8524 <genNoiseCoron>
   0x000085b4 <+24>: ldr      r3, [pc, #40] ; <noiseCoron+72>
   0x000085b8 <+28>: ldr      r2, [r4]
   0x000085bc <+32>: ldr      r1, [r3, r2, lsl #2]
   0x000085c0 <+36>: mov      r3, #0
   0x000085c4 <+40>: cmp      r3, r1
   0x000085c8 <+44>: beq      0x85d4 <noiseCoron+56>
   0x000085cc <+48>: add      r3, r3, #1
   0x000085d0 <+52>: b        0x85c4 <noiseCoron+40>
   0x000085d4 <+56>: add      r2, r2, #1
   0x000085d8 <+60>: str      r2, [r4]
   0x000085dc <+64>: pop      {r4, pc}
   0x000085e0 <+68>: andeq    r0, r1, r0, asr r8
   0x000085e4 <+72>: andeq    r0, r1, r8, asr r8
End of assembler dump.
```

Random precharging

Protection against power analysis in a Hamming Distance model

- Example: Leakage on value *v* is charged in memory or in a register:

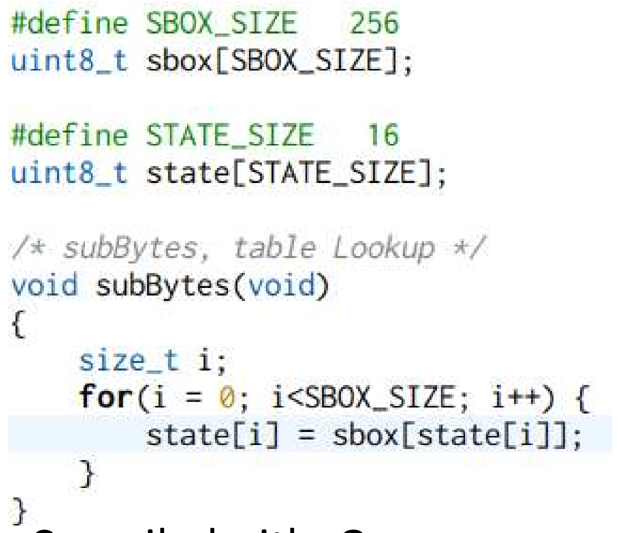
```
    insn_k
#1  mem <- v

    insn_k
#2  reg <- v
```

Leakage: HD(v,k)

- Random precharging: the variable assignment is preceded by an assignment using a **mask *m***, unknown to the attacker:

```
    insn_k
    mem <- m
#1  mem <- v

    insn_k
    reg <- m
#2  reg <- v
```

Leakage:
HD(v,m) = HW(v+m)

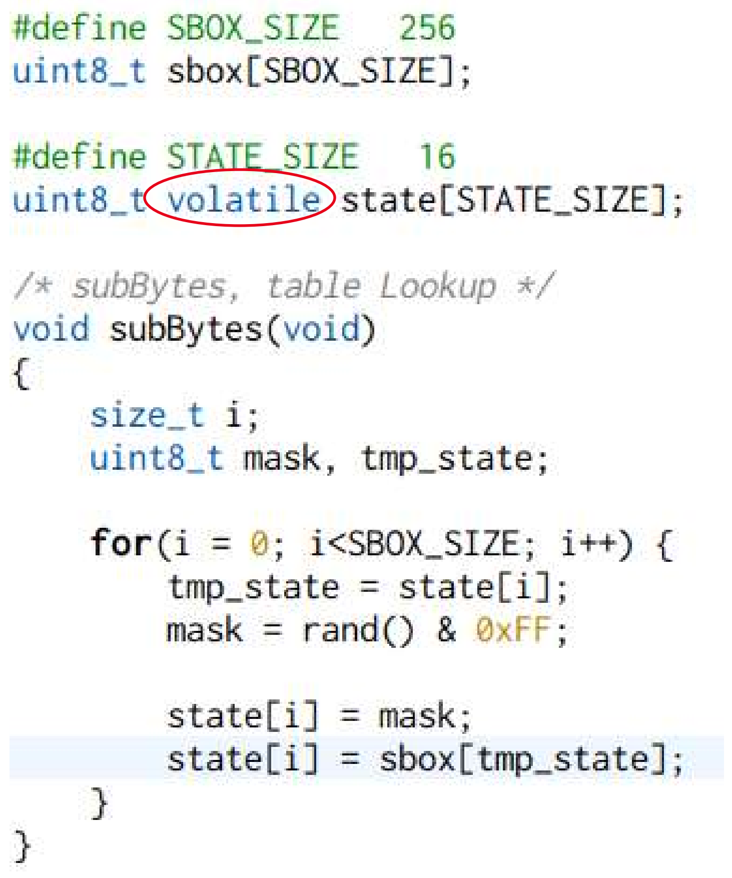


```
#define SBOX_SIZE   256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE   16
uint8_t state[STATE_SIZE];

/* subBytes, table Lookup */
void subBytes(void)
{
    size_t i;
    for(i = 0; i<SBOX_SIZE; i++) {
        state[i] = sbox[state[i]];
    }
}
```

Compiled with -Os:

```
0x0000 <+0>:  mov r3, #0
0x0004 <+4>:  ldr r2, [pc, #28] ; 0x28 <subBytes+40>
0x0008 <+8>:  ldr r0, [pc, #28] ; 0x2c <subBytes+44>
0x000c <+12>: ldrb r1, [r3, r2]
0x0010 <+16>: ldrb r1, [r0, r1]
0x0014 <+20>: strb r1, [r3, r2] ; leaky instruction
0x0018 <+24>: add r3, r3, #1
0x001c <+28>: cmp r3, #16
0x0020 <+32>: bne 0xc <subBytes+12>
0x0024 <+36>: bx lr
0x0028 <+40>: andeq r0, r0, r0
0x002c <+44>: andeq r0, r0, r0
```

Random precharging

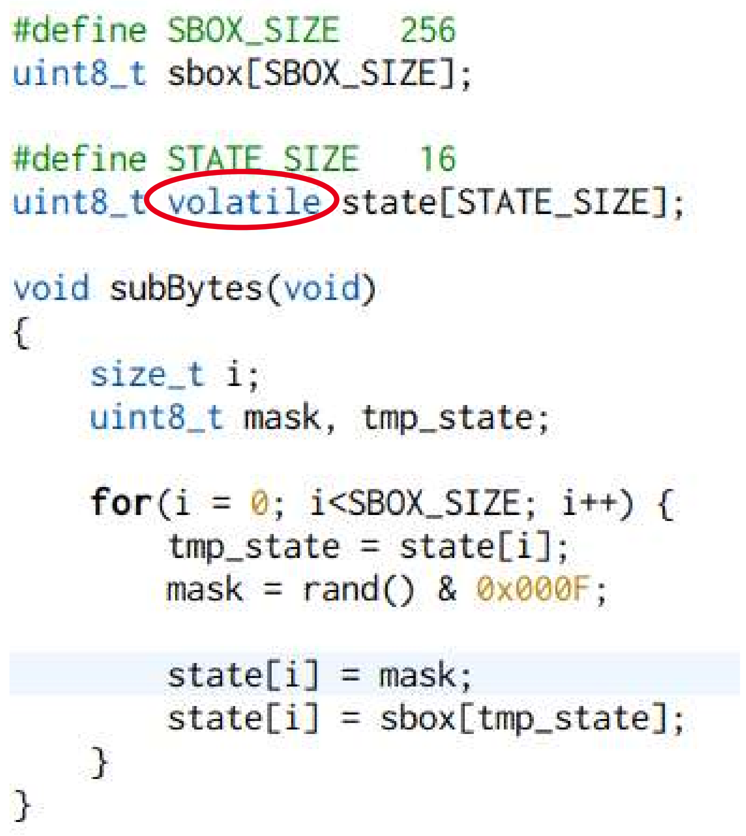


```c
#define SBOX_SIZE   256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE   16
uint8_t volatile state[STATE_SIZE];

/* subBytes, table Lookup */
void subBytes(void)
{
    size_t i;
    uint8_t mask, tmp_state;

    for(i = 0; i<SBOX_SIZE; i++) {
        tmp_state = state[i];
        mask = rand() & 0xFF;

        state[i] = mask;
        state[i] = sbox[tmp_state];
    }
}
```

Compiled with -Os:

```
0x0000 <+0>:  push {r4, r5, r6, r7, r8, lr}
0x0004 <+4>:  mov r4, #0
0x0008 <+8>:  ldr r5, [pc, #48] ; <subBytes+64>
0x000c <+12>: ldr r7, [pc, #48] ; <subBytes+68>
0x0010 <+16>: ldrb r6, [r5, r4]
0x0014 <+20>: bl <rand>
0x0018 <+24>: and r6, r6, #255 ; 0xff
0x001c <+28>: ldrb r3, [r7, r6]
0x0020 <+32>: and r0, r0, #15
0x0024 <+36>: strb r0, [r5, r4]
0x0028 <+40>: strb r3, [r5, r4]
0x002c <+44>: add r4, r4, #1
0x0030 <+48>: cmp r4, #16
0x0034 <+52>: bne 0x10 <subBytes+16>
0x0038 <+56>: pop {r4, r5, r6, r7, r8, lr}
0x003c <+60>: bx lr
0x0040 <+64>: andeq r0, r0, r0
0x0044 <+68>: andeq r0, r0, r0
```

Random precharging



```
#define SBOX_SIZE   256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE   16
uint8_t volatile state[STATE_SIZE];

void subBytes(void)
{
    size_t i;
    uint8_t mask, tmp_state;

    for(i = 0; i<SBOX_SIZE; i++) {
        tmp_state = state[i];
        mask = rand() & 0x000F;

        state[i] = mask;
        state[i] = sbox[tmp_state];
    }
}
```

Compiled with -O1:

```
0x0000 <+0>:  push {r4, r5, r6, r7, r8, lr}
0x0004 <+4>:  mov r4, #0
0x0008 <+8>:  ldr r6, [pc, #48] ; <subBytes+64>
0x000c <+12>: ldr r7, [pc, #48] ; <subBytes+68>
0x0010 <+16>: ldrb r5, [r6, r4]
0x0014 <+20>: and r5, r5, #255 ; 0xff
0x0018 <+24>: bl <rand>
0x001c <+28>: and r0, r0, #15
0x0020 <+32>: strb r0, [r6, r4]
0x0024 <+36>: ldrb r3, [r7, r5]
0x0028 <+40>: strb r3, [r6, r4]
0x002c <+44>: add r4, r4, #1
0x0030 <+48>: cmp r4, #16
0x0034 <+52>: bne 0x10 <subBytes+16>
0x0038 <+56>: pop {r4, r5, r6, r7, r8, lr}
0x003c <+60>: bx lr
0x0040 <+64>: andeq r0, r0, r0
0x0044 <+68>: andeq r0, r0, r0
```

Instruction reordering introduces leakage

Huh??

So…

Let’s avoid compiler optimisations!

Compiling with -O0

- All program variables are moved onto the stack before anything else
- Register spilling (> -O0): the register value is moved to the stack

  => Information leakage!

- Bigger code size -> larger attack surface

  => More potential vulnerabilies

```
Dump of assembler code for function subBytes:
0x84e4 <+0>:  push {r11}     ; (str r11, [sp, #-4]!)
0x84e8 <+4>:  add  r11, sp, #0
0x84ec <+8>:  sub  sp, sp, #12
0x84f0 <+12>: mov  r3, #0
0x84f4 <+16>: str  r3, [r11, #-8]
0x84f8 <+20>: b       0x8530 <subBytes+76>
0x84fc <+24>: ldr  r2, [pc, #68] ; <subBytes+100>
0x8500 <+28>: ldr  r3, [r11, #-8]
0x8504 <+32>: add  r3, r2, r3
0x8508 <+36>: ldrb r3, [r3]
0x850c <+40>: ldr  r2, [pc, #56] ; <subBytes+104>
0x8510 <+44>: ldrb r2, [r2, r3]
0x8514 <+48>: ldr  r1, [pc, #44] ; <subBytes+100>
0x8518 <+52>: ldr  r3, [r11, #-8]
0x851c <+56>: add  r3, r1, r3
0x8520 <+60>: strb r2, [r3]
0x8524 <+64>: ldr  r3, [r11, #-8]
0x8528 <+68>: add  r3, r3, #1
0x852c <+72>: str  r3, [r11, #-8]
0x8530 <+76>: ldr  r3, [r11, #-8]
0x8534 <+80>: cmp  r3, #15
0x8538 <+84>: bls  0x84fc <subBytes+24>
0x853c <+88>: sub  sp, r11, #0
0x8540 <+92>: pop  {r11} ; (ldr r11, [sp], #4)
0x8544 <+96>: bx   lr
```

Securing compilation

Compilation of a Countermeasure Against Instruction-Skip Faults

Fault model: **instruction skips** [Moro et al., 2014]

Can be protected by the **duplication of idempotent instructions** [Moro et al., 2014]

```
cmp r0, #1
```

idempotent

```
cmp r0, #1
cmp r0, #1
```

```
add r1, r1, r2
```

non-idempotent

mv r_{new}, r1
mv r_{new}, r1
add r1, r_{new}, r2
add r1, r_{new}, r2

push {r4, r7, lr}
add r7, sp, #4
...
bl byte_compare
~~cmp r0, #1~~ nop
bne .BB2
b .BB1

[Moro, 2014] Formal verification of a software countermeasure against instruction skip attacks. doiXXX
[Barry, 2016] Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. doiXXX

Compilation of a Countermeasure Against Instruction-Skip Faults

Fault model: **instruction skips** [Moro et al., 2014]

Can be protected by the **duplication of idempotent instructions** [Moro et al., 2014]

```
cmp r0, #1
```

idempotent

```
cmp r0, #1
cmp r0, #1
```

```
add r1, r2, r3
```

idempotent

```
add r1, r2, r3
add r1, r2, r3
```

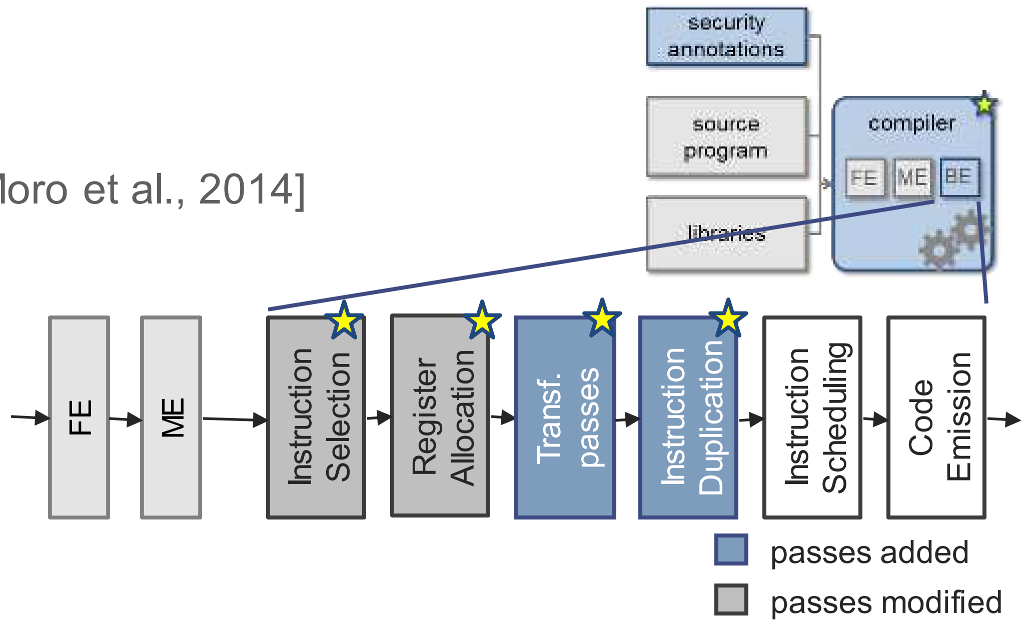
```
ldr r3, [r1, #4]
ldr r3, [r1, #4]
add r0, r1, r2
add r0, r1, r2
```

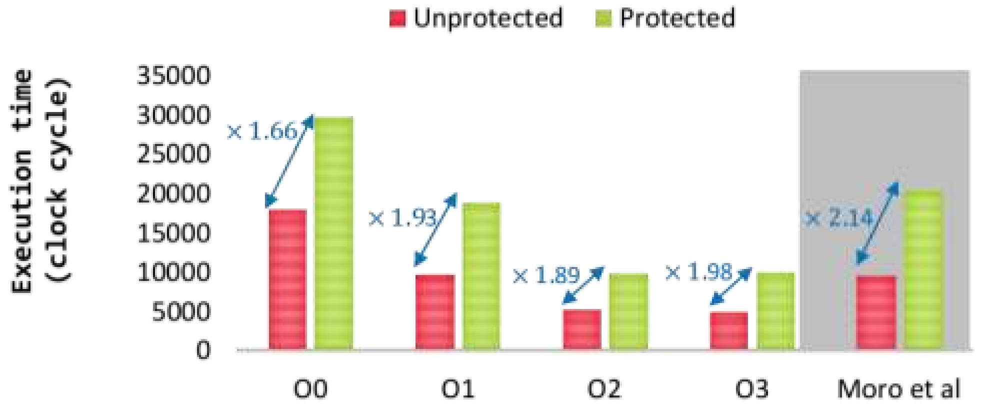
(6 cycles)

instruction scheduling

```
ldr r3, [r1, #4]
add r0, r1, r2
ldr r3, [r1, #4]
add r0, r1, r2
```

(5-6 cycles)





Compilation of a Countermeasure Against Instruction-Skip Faults

Fault model: **instruction skips** [Moro et al., 2014]

Can be protected by the **duplication of idempotent instructions** [Moro et al., 2014]

```
cmp r0, #1
```

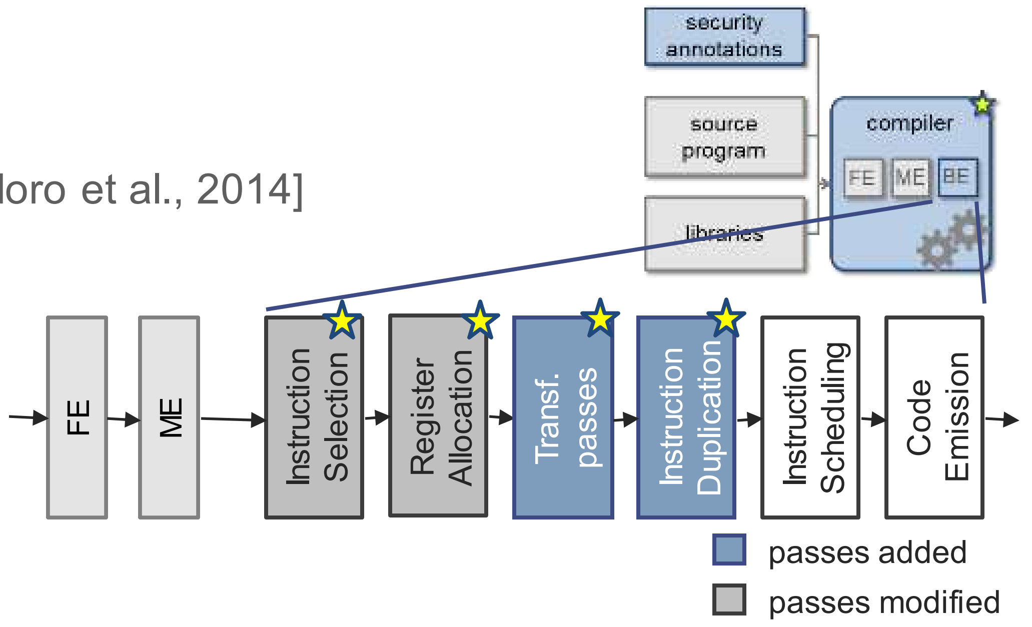
idempotent

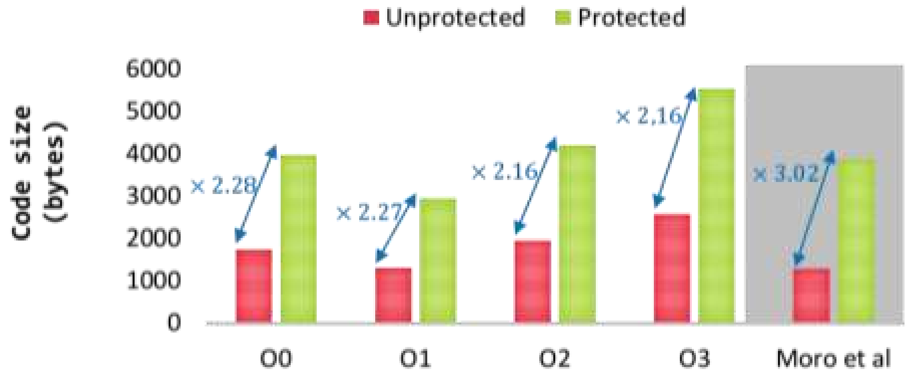
```
cmp r0, #1
cmp r0, #1
```

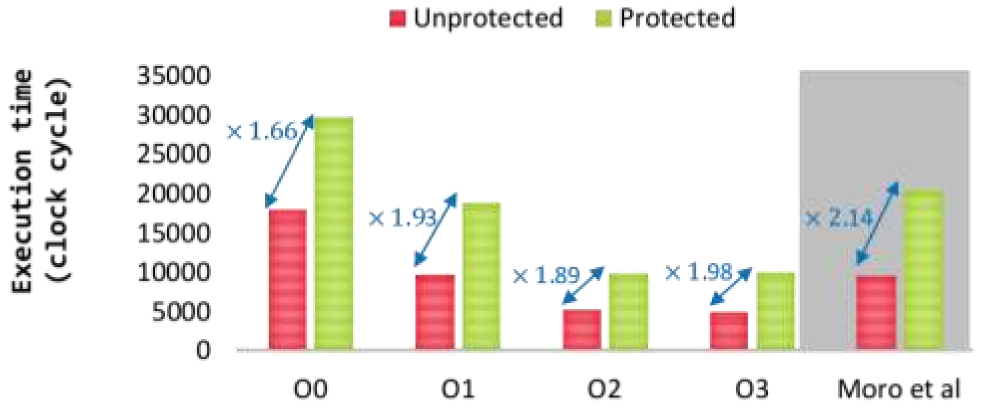
```
add r1, r2, r3
```

idempotent

```
add r1, r2, r3
add r1, r2, r3
```

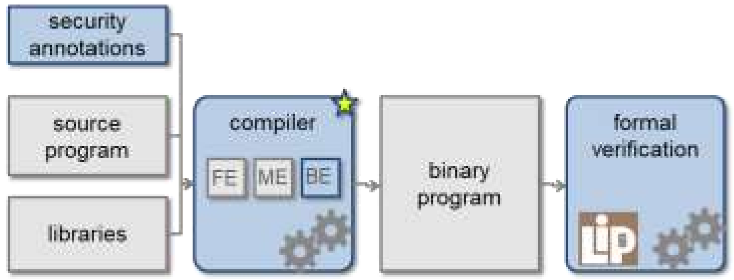






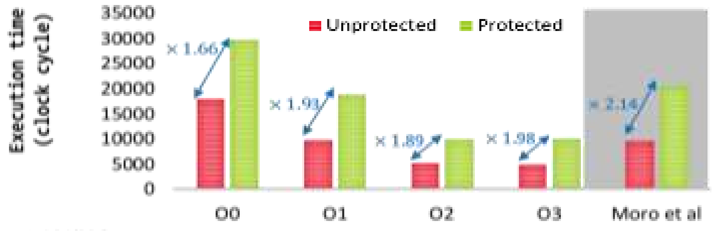
[Moro, 2014] Formal verification of a software countermeasure against instruction skip attacks. doiXXX
[Barry, 2016] Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. doiXXX

Compilation of a Countermeasure Against Instruction-Skip Faults



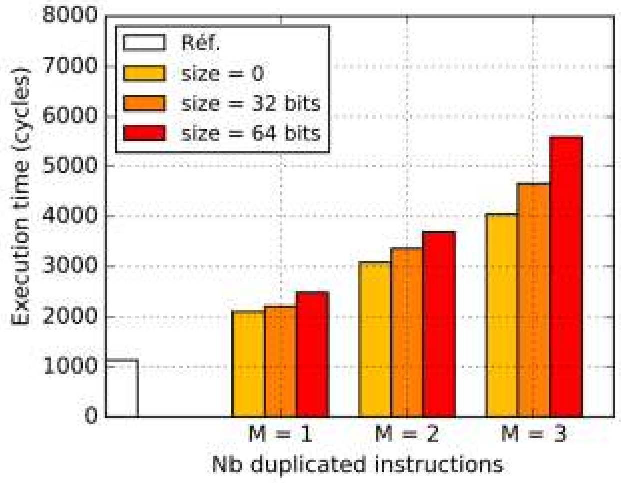
Experimental results:

- Reduced overhead wrt. original implementation (execution time, code size)
- Generalisation of the protection scheme, supported by compilation parameters:
  - **M**: Nb of faults
  - **size**: Fault width
  - **Target functions**
- Fine-grained countermeasure application, to selected functions, reduces the execution overhead below ×1.23 and size overheads below ×1.12 [Barry, 2017]
- Formally verified by RobustB / LIP6 (IDROMEL project) [Belleville, 2021]

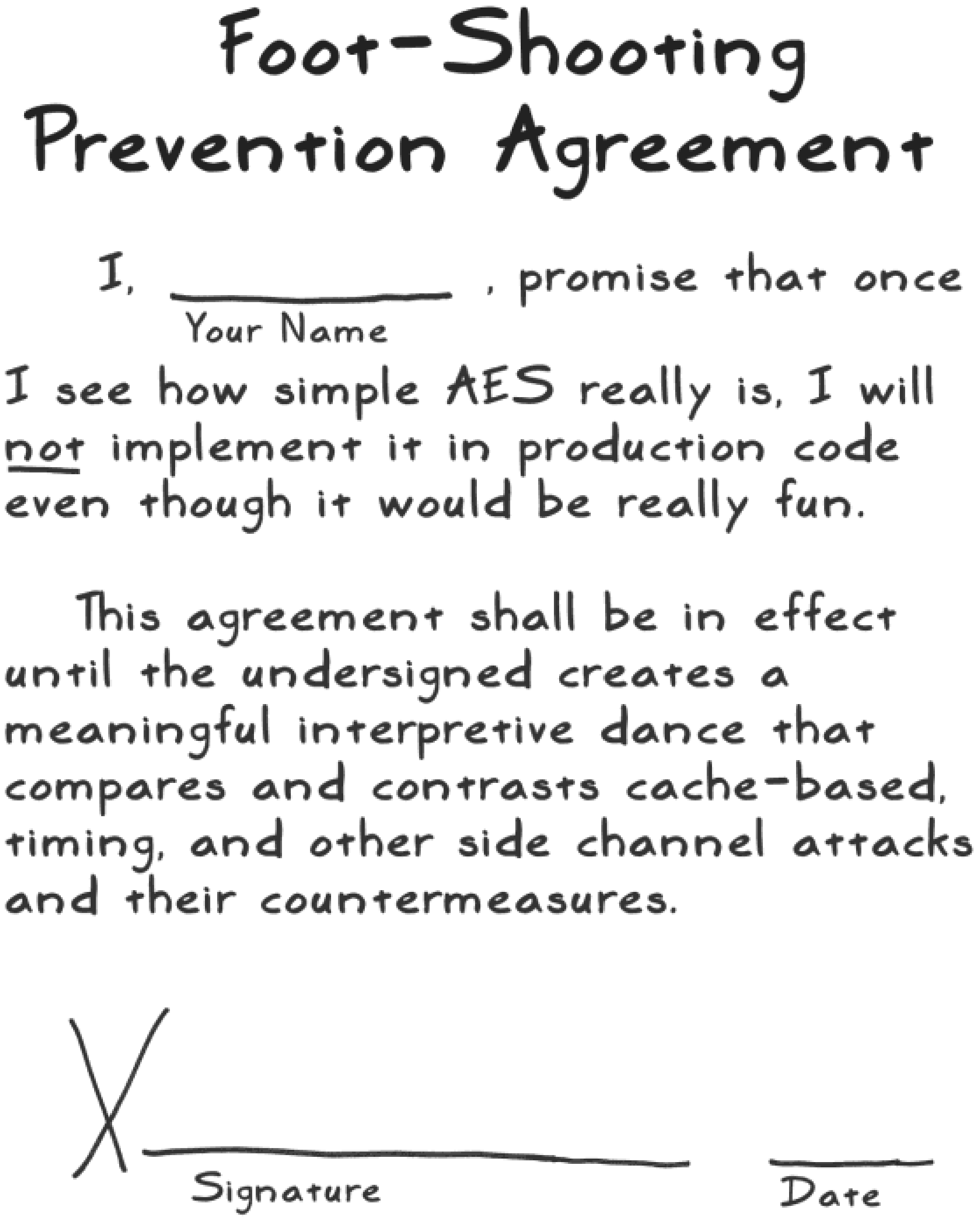


But not effective in practice!

- Experimental evaluation on a **laser bench**
- The target platform (STM32) is **intrinsically vulnerable** to laser FI
- Thin fault coverage:
  - protects instructions only against skip, but not against byte corruption (hypothetical fault model);
  - no data protection
- Software countermeasures, since not effective, increase the attack surface and often lead to easier exploitation of fault injection.



[Moro, 2014] Formal verification of a software countermeasure against instruction skip attacks. doiXXX
[Barry, 2016] Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. doiXXX
[Barry, 2017] XXX
[Belleville, 2021] XXX



Foot-Shooting Prevention Agreement

I, ______________ , promise that once
Your Name
I see how simple AES really is, I will ***not*** implement it in production code even though it would be really fun.

This agreement shall be in effect until the undersigned creates a meaningful interpretive dance that compares and contrasts cache-based, timing, and other side channel attacks and their countermeasures.

X____________________ ________
Signature Date