



RÉPUBLIQUE
FRANÇAISE

*Liberté
Égalité
Fraternité*



Pew Pew, I'm root!

De la caractérisation à l'exploitation : un voyage plein d'embûche

Alexandre Iooss, Thomas Trouchkine et **Guillaume Bouffard**
Agence nationale de la sécurité des systèmes d'information

Les attaques par perturbation



Pour réussir une attaque par perturbation, il est nécessaire de :

- Savoir **où** attaquer? Position (x, y, z, θ) sur le composant cible.
- Savoir **comment** attaquer? \Rightarrow Paramètres du médium d'injection.
- Savoir **quand** attaquer? \Rightarrow L'instant t du programme cible.

Les attaques par perturbation (cont.)



- 1 **Où ?/Comment ?** : Caractériser la sensibilité du composant cible à un médium de faute [TBC21, PHB⁺19]
- 2 **Quand ?** : Transposer cette sensibilité à un applicatif cible [GHHR23, Dur16]
- 3 **Réaliser son attaque via une exploitation** [Wer22, TBE⁺21]



- 1 **Où ?/Comment ?** : Caractériser la sensibilité du composant cible à un médium de faute [TBC21, PHB⁺19]
- 2 **Quand ?** : Transposer cette sensibilité à un applicatif cible [GHHR23, Dur16]
- 3 Réaliser son attaque via une exploitation [Wer22, TBE⁺21]

Problématique

Peut-on trouver **quand** fauter un logiciel complexe en caractérisant (**où/comment**) l'effet d'une faute ?



- 1 **Où ?/Comment ?** : Caractériser la sensibilité du composant cible à un médium de faute [TBC21, PHB⁺19]
- 2 **Quand ?** : Transposer cette sensibilité à un applicatif cible [GHHR23, Dur16]
- 3 Réaliser son attaque via une exploitation [Wer22, TBE⁺21]

Problématique

Peut-on trouver **quand** fauter un logiciel complexe en caractérisant (**où/comment**) l'effet d'une faute ?

Cible : l'application sudo sur un Raspberry Pi 4 sous Raspberry Pi OS¹ (dérivée de Debian).

1. Voir : <https://www.raspberrypi.com/software/>

1. Caractérisation

Analyse des effets des fautes sur un CPU complexe [Tro21]



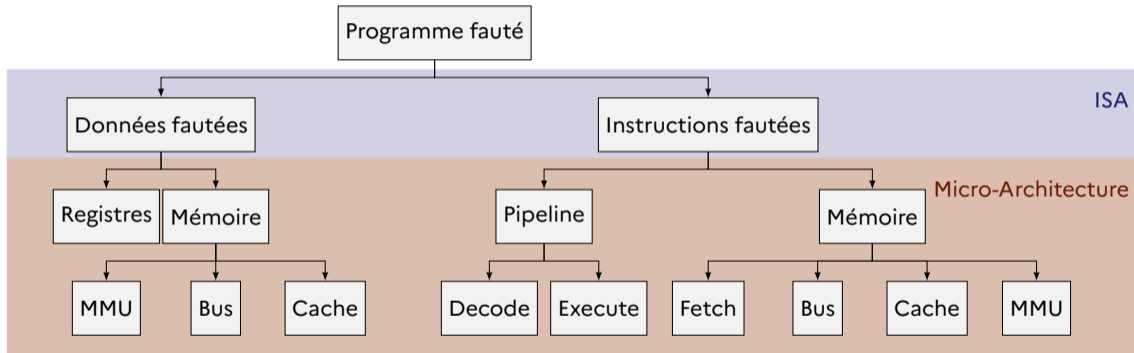
Durant une faute, au moins un bloc de micro-architecture est perturbé.



Durant une faute, au moins un bloc de micro-architecture est perturbé.

Modus operandi

- Un programme de test est fauté durant son exécution
- Fauter différents programmes de tests pour obtenir des informations sur le comportement des blocs micro-architecturaux.



Programme de test 1

```
orr r4, r4;
```

```
/*
```

```
* Nombre arbitraire  
* de répétitions
```

```
*/
```

```
orr r4, r4;
```

Programme de test 2

```
and r4, r4, #255;
```

```
/*
```

```
* Nombre arbitraire  
* de répétitions
```

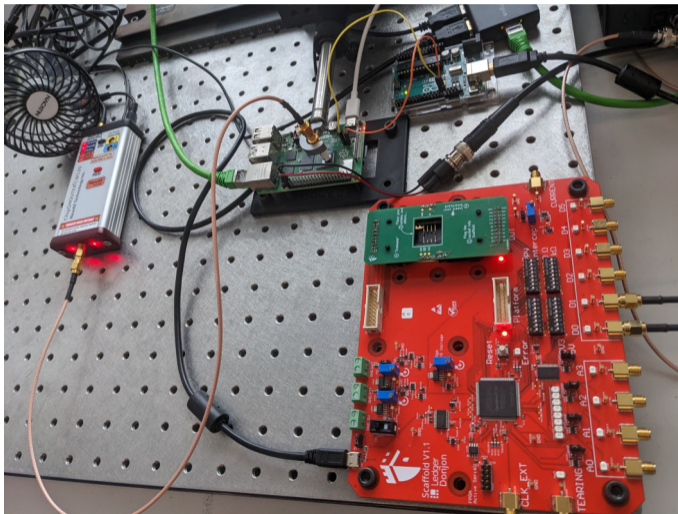
```
*/
```

```
and r4, r4, #255;
```

Table – Valeurs initiales des registres.

Registre	Valeur initiale pour la répétition ORR R4,R4	Valeur initiale pour répétition AND R4,#255
r0	0xFFFE0001	0xFFFE0001
r1	0xFFFD0002	0xFFFD0002
r2	0xFFFB0004	0xFFFB0004
r3	0xFFF70008	0x000000FF
r4	0xFFEF0010	0xFFEF0010
r5	0xFFDF0020	0xFFDF0020
r6	0xFFBF0040	0xFFBF0040
r7	0xFF7F0080	0xFF7F0080
r8	0xFEFF0100	0xFEFF0100
r9	0xFDFF0200	0xFDFF0200

Banc d'essai



Carte de sensibilité et résultats

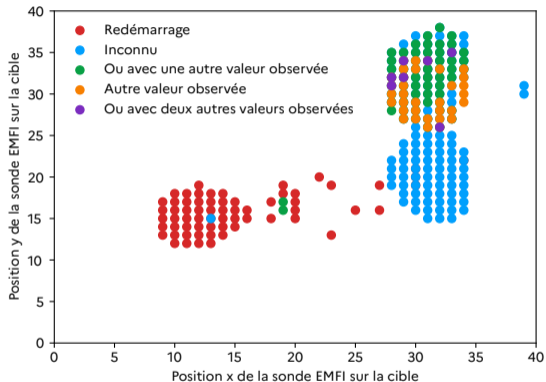


Figure – ORR R4, R4

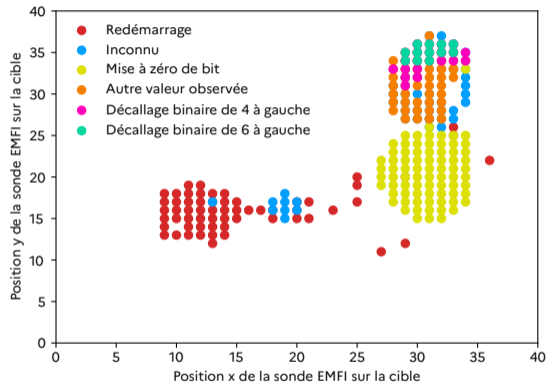


Figure – AND R4, R4, #255

Carte de sensibilité et résultats

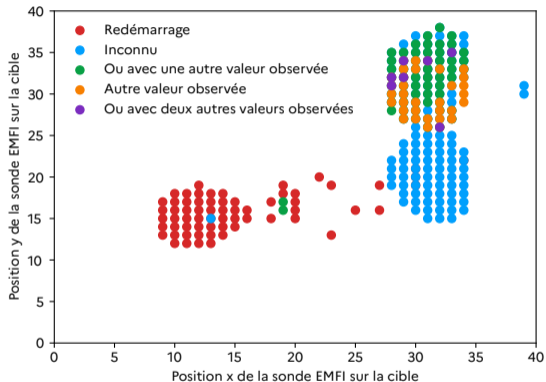


Figure – ORR R4, R4

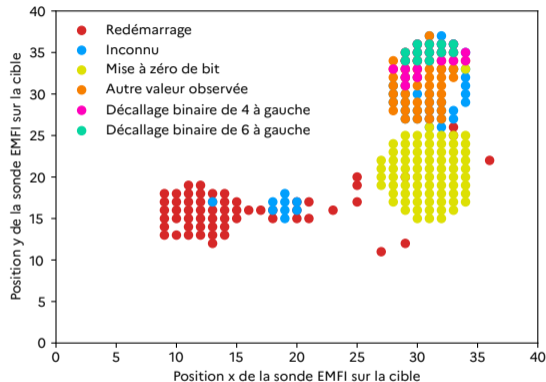


Figure – AND R4, R4, #255

Mise à 0 des 8 bits de poids faible dans les instructions exécutées

2. Transposition du modèle de faute



But de la simulation : savoir **quand** injecter la faute dans le flot d'exécution pour obtenir un effet désiré. On se limite à du simple-faute.

Cible : binaire sudo de Raspberry Pi OS² (basé sur Debian)

2. https://downloads.raspberrypi.org/raspios_lite_armhf/images/raspios_lite_armhf-2022-09-26/

But de la simulation : savoir **quand** injecter la faute dans le flot d'exécution pour obtenir un effet désiré. On se limite à du simple-faute.

Cible : binaire sudo de Raspberry Pi OS² (basé sur Debian)

Règles d'authentification dans /etc/pam.d/common-auth

```
# here are the per-package modules (the "Primary" block)
auth          [success=1 default=ignore]      pam_unix.so nullok
# here's the fallback if no module succeeds
auth          requisite                       pam_deny.so
```

sudo appelle PAM qui utilise le module pam_unix.so.

2. https://downloads.raspberrypi.org/raspios_lite_armhf/images/raspios_lite_armhf-2022-09-26/



Rainbow est un simulateur d'injections de faute écrit par le Donjon de Ledger, basé sur Unicorn-Engine (QEMU).

⚠ sudo est lié **dynamiquement** avec la glibc et PAM.



Rainbow est un simulateur d'injections de faute écrit par le Donjon de Ledger, basé sur Unicorn-Engine (QEMU).

⚠ sudo est lié **dynamiquement** avec la glibc et PAM.

On a implémenté un chargeur basé sur CLE dans Rainbow.

Dépendances dynamiques de pam_unix.so

```
libpam.so.0, libcrypt.so.1, libselinux.so.1, libnsl.so.2,  
libtirpc.so.3, libc.so.6, ld-linux-armhf.so.3, libaudit.so.1,  
libdl.so.2, libpcre2-8.so.0, libgssapi_krb5.so.2,  
libpthread.so.0, libcap-ng.so.0, libkrb5.so.3,  
libk5crypto.so.3, libcom_err.so.2, libkrb5support.so.0,  
libkeyutils.so.1, libresolv.so.2
```



 Ledger-Donjon/rainbow



```
1 def fault_model(emu):
2     # Get PC value
3     pc = emu["pc"]
4     # Get next instruction
5     instr = emu[pc]
6
7     # Patch and run modified instruction
8     i = int.from_bytes(instr, "little") & 0xFFFF_FF00
9     instr_patched = i.to_bytes(4, "little")
10    emu[pc] = instr_patched
11    emu.start(pc, 0, count=1)
12
13    # Restore correct instruction
14    emu[pc] = bytes(instr)
```

Test exhaustif en simple-faute



```
1  emu = rainbow_arm()
2  emu.load("arm-libs/pam_unix.so")
3  emu[0xE0000000] = f"toto\x00".encode()
4  emu[0xF0000000] = f"$6$hH.15uU5laaxuXHY$anemvMyc.gFyc[...]nSGEO.\x00".encode()
5  emu["r0"] = 0 # pamh (used for pam_syslog calls)
6  emu["r1"] = 0xE0000000 # const char *p
7  emu["r2"] = 0xF0000000 # char *hash
8  emu["r3"] = 1 # unsigned int nullok
9  pc_stopped = emu.start_and_fault(fault_model, i, 0x00405b40, 0, count=1000)
10 print(emu["r0"]) # if 0, then auth is successful
```

Variante de l'article de blog du Donjon [IS22]. On hook malloc, calloc et free.
22 minutes ou 3.2 secondes avec SHA2-256 hooké (crypt_r).

Test exhaustif en simple-faute : résultats



Faute trouvée

```

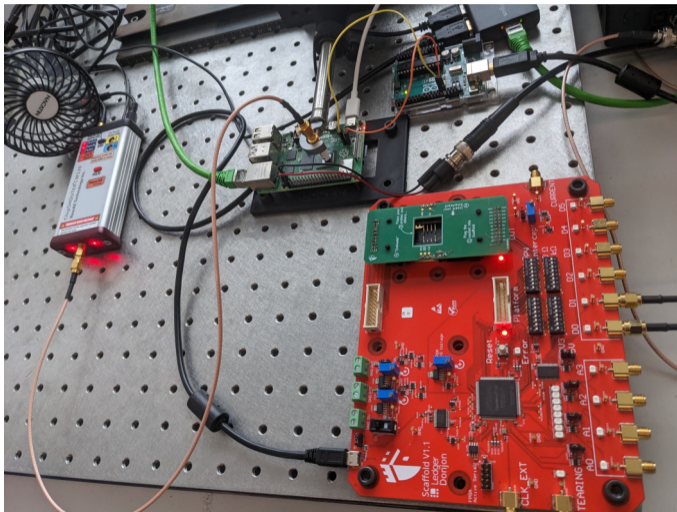
; i=1201 (in pam_unix.so)
MOVNE R4, #7
; become
MOVNE R4, #0
    
```

```

; [...]
movne r4,#0x7 ; r4 <- 0 with fault
cmp r3,#0x0
cpyne r3,r7
movne r1,#0x0
beq LAB_00405cb4
LAB_00405ca4:
strb r1,[r3],#0x1
ldrb r2,[r3,#0x0]
cmp r2,#0x0
bne LAB_00405ca4
LAB_00405cb4:
cpy r0,r7
bl free
cpy r0,r4 ; return the value of r4
ldmia sp!,{r4,r5,r6,r7,r8,r9,r10,pc}
    
```

3. Exploitation

Banc d'essai



Passage à la réalité



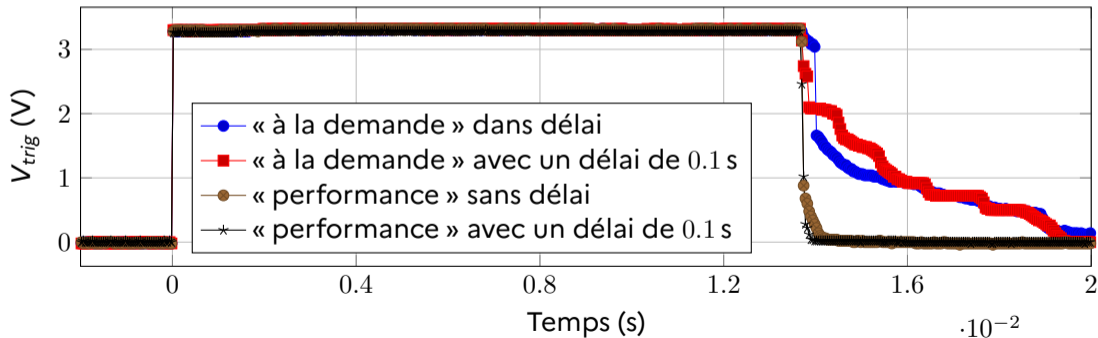


Modus operandi

- Création d'un wrapper pour lancer sudo via Netcat et trigger un GPIO.
- sudo forcé sur le coeur 3 du CPU.

Modus operandi

- Création d'un wrapper pour lancer sudo via Netcat et trigger un GPIO.
- sudo forcé sur le coeur 3 du CPU.



Astuce pour retrouver l'instant de faute



Utilisation d'un open-sample

The image displays a debugger window with two views: assembly and decompiled code.

Assembly View (Left):

```

LAB_00405c7c
00405c7c 05 10 a0 e1 cpy     r1,r5
00405c80 07 00 a0 e1 cpy     r0,r7
00405c84 96 ee ff eb bl      <EXTERNAL>::strcmp
00405c88 00 30 d7 e5 ldrb   r3,[r7,#0x0]
00405c8c 00 40 50 e2 subs   r4,r0,#0x0
00405c90 02 00 a0 e3 mov     r0,#0x2
00405c94 01 20 a0 e3 mov     r2,#0x1
00405c98 04 70 a0 e3 mov     r7,#0x4
00405c9c 00 00 00 ef swi    0x0
00405ca0 03 00 00 0a beq    LAB_00405cb4

LAB_00405ca4
00405ca4 01 10 c3 e4 strb   r1,[r3],#0x1
00405ca8 00 20 d3 e5 ldrb   r2,[r3,#0x0]
00405cac 00 00 52 e3 cmp     r2,#0x0
00405cb0 fb ff ff 1a bne    LAB_00405ca4

LAB_00405cb4
00405cb4 07 00 a0 e1 cpy     r0,r7
  
```

Assembly View (Right):

```

LAB_00405c7c
00405c7c 05 10 a0 e1 cpy     r1,r5
00405c80 07 00 a0 e1 cpy     r0,r7
00405c84 96 ee ff eb bl      libc.so.6::strcmp
00405c88 00 30 d7 e5 ldrb   r3,[r7,#0x0]
00405c8c 00 40 50 e2 subs   r4,r0,#0x0
00405c90 07 40 a0 13 movne  r4,#0x7
00405c94 00 00 53 e3 cmp     r3,#0x0
00405c98 07 30 a0 11 cpyne  r3,r7
00405c9c 00 10 a0 13 movne  r1,#0x0
00405ca0 03 00 00 0a beq    LAB_00405cb4

LAB_00405ca4
00405ca4 01 10 c3 e4 strb   r1,[r3],#0x1
00405ca8 00 20 d3 e5 ldrb   r2,[r3,#0x0]
00405cac 00 00 52 e3 cmp     r2,#0x0
00405cb0 fb ff ff 1a bne    LAB_00405ca4

LAB_00405cb4
00405cb4 07 00 a0 e1 cpy     r0,r7
  
```

Decompiled Code View (Right):

```

120     *pbVar8 = 0;
121     bVar1 = pbVar8[1];
122     pbVar8 = pbVar8 + 1;
123   }
124   }
125 }
126 iVar4 = strcmp((char *)pbVar9,hash);
127 pcVar3 = (char *) (uint)*pbVar9;
128 bVar10 = iVar4 == 0;
129 /* can be faulted with 0x0FFF_FF00 */
130 cVar5 = extraout_r1_00;
131 LAB_00405c94:
132 software_interrupt(0);
133 if (!bVar10) {
134   do {
135     pcVar7 = pcVar3 + 1;
136     *pcVar3 = cVar5;
137     pcVar3 = pcVar7;
138   } while (*pcVar7 != '\0');
139 }
140 free((void *)0x4);
141 return iVar4;
142 }
143 }
  
```

Problème : le délais introduit par le `syscall` et processus parent.

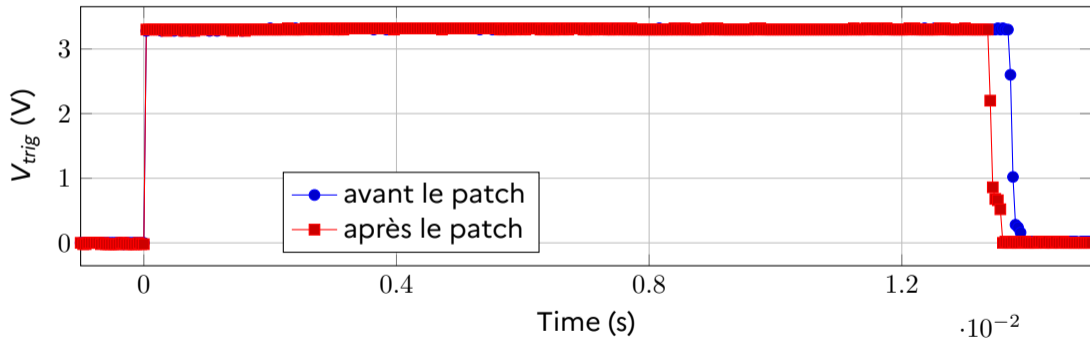





Figure – Moyen de 1024 appels de `sudo` avec le mauvais mot de passe, CPU mode performance, délai de 0.1 s.



Dans ce travail :

- Transposition du modèle de faute sur un programme complexe (sudo)
- ⚠ Passage d'une analyse d'un binaire à une attaque sur banc
- Usage d'outils open-source :
 -  Ledger-Donjon/rainbow (proposition de patchs soumis).
 -  Ledger-Donjon/scaffold
 -  newaetech/ChipSHOUTER



Dans ce travail :

- Transposition du modèle de faute sur un programme complexe (sudo)
- ⚠ Passage d'une analyse d'un binaire à une attaque sur banc
- Usage d'outils open-source :
 - 🔄 Ledger-Donjon/rainbow (proposition de patchs soumis).
 - 🔄 Ledger-Donjon/scaffold
 - 🔄 newaetech/ChipSHOUTER

et pour la suite ?

- Peut-on se passer d'un *open-sample* ?
- Comment s'en protéger ?
- Passage au RISC-V.



Pew Pew, I'm root!

De la caractérisation à l'exploitation : un voyage plein d'embûche

Alexandre Iooss

Laboratoire Sécurité des Composants

ANSSI

`alexandre.iooss@ssi.gouv.fr`

Guillaume Bouffard

Laboratoire Architectures Matérielles et
Logicielles

ANSSI

`guillaume.bouffard@ssi.gouv.fr`



- [Dur16] Louis Dureuil, *Analyse de code et processus d'évaluation des composants sécurisés contre l'injection de faute.*, Ph.D. thesis, Grenoble Alpes University, France, 2016.
- [GHHR23] Antoine Gicquel, Damien Hardy, Karine Heydemann, and Erven Rohou, *SAMVA : static analysis for multi-fault attack paths determination*, Constructive Side-Channel Analysis and Secure Design - 14th International Workshop, COSADE 2023, Munich, Germany, April 3-4, 2023, Proceedings (Elif Bilge Kavun and Michael Pehl, eds.), Lecture Notes in Computer Science, vol. 13979, Springer, 2023, pp. 3–22.
- [IS22] Alexandre Iooss and Victor Servant, *Integrating fault injection in development workflows*, Aug 2022.



- [PHB⁺19] Julien Proy, Karine Heydemann, Alexandre Berzati, Fabien Majéric, and Albert Cohen, *A first isa-level characterization of EM pulse effects on superscalar microarchitectures : A secure software perspective*, Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019, ACM, 2019, pp. 7:1–7:10.
- [TBC21] Thomas Trouchkine, Guillaume Bouffard, and Jessy Clédière, *EM fault model characterization on socs : From different architectures to the same fault model*, 18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021, IEEE, 2021, pp. 31–38.
- [TBE⁺21] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard, *Electromagnetic fault injection against a complex cpu, toward new micro-architectural fault models*, J. Cryptogr. Eng. **11** (2021), no. 4, 353–367.



- [Tro21] Thomas Trouchkine, *SoC physical security evaluation*, Theses de doctorat, Université Grenoble Alpes, March 2021.
- [Wer22] Vincent Werner, *Optimizing identification and exploitation of fault injection vulnerabilities on microcontrollers.*, Ph.D. thesis, Grenoble Alpes University, France, 2022.