Software Fault Injection for Software Certification

Roberto Natella
Critiware s.r.l.

9th Automotive SPIN Italy Workshop
Milan, December 1, 2011

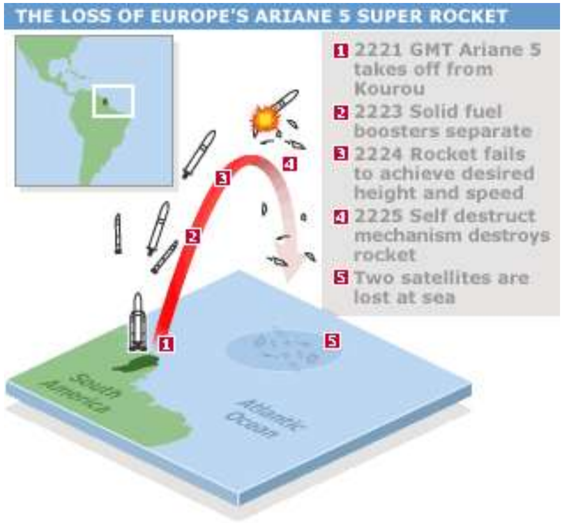
AUTOMOTIVE SPIN ITALIA

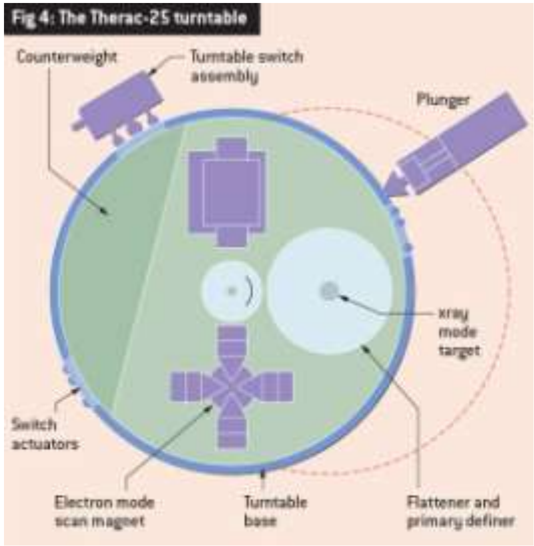
CRITIWARE s.r.l.
Innovating Critical Systems Engineering

Safety-critical software

CRITIWARE s.r.l.
Innovating Critical Systems Engineering

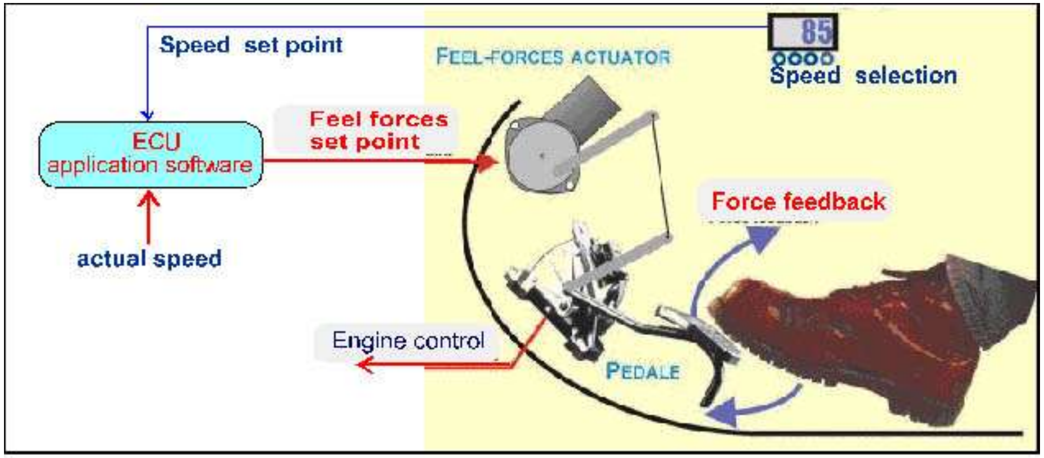
- Unfortunately, it is practically impossible to guarantee that software is defect-free
    - Complexity
    - Time-to-market constraints
- Many accidents due to "well-tested" software



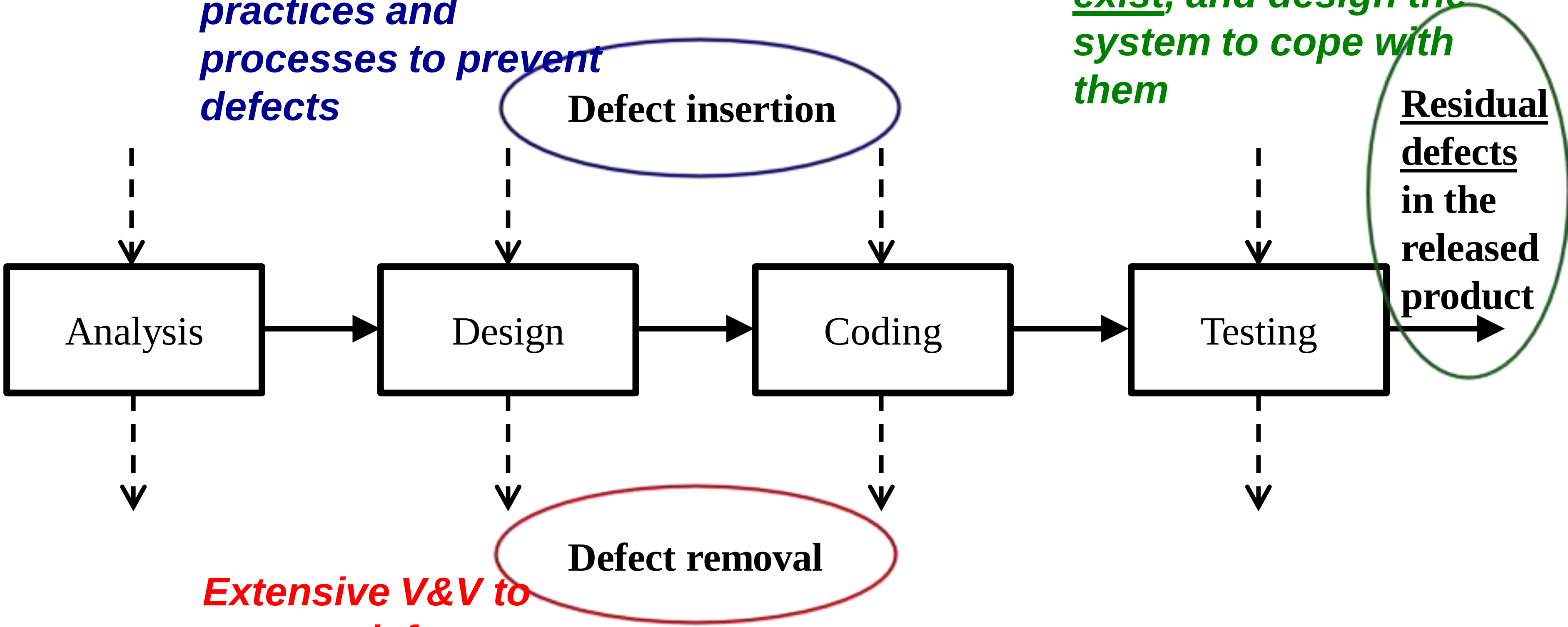


The Toyota software failure

- Due to a software defect, Toyota recalled almost half a million new cars
- The issue causes the *unintended acceleration* of the vehicle
- Numerous investigations have taken place (also by the NASA JPL laboratory), but the causes of the problem are **still unclear after several months**



Dealing with software faults



Rigorous development practices and processes to prevent defects

<u>Assume that residual defects do exist</u>, and design the system to cope with them

Defect insertion

Analysis

Design

Coding

Testing

<u>Residual defects</u> in the released product

Defect removal

Extensive V&V to remove defects (more and more testing...)

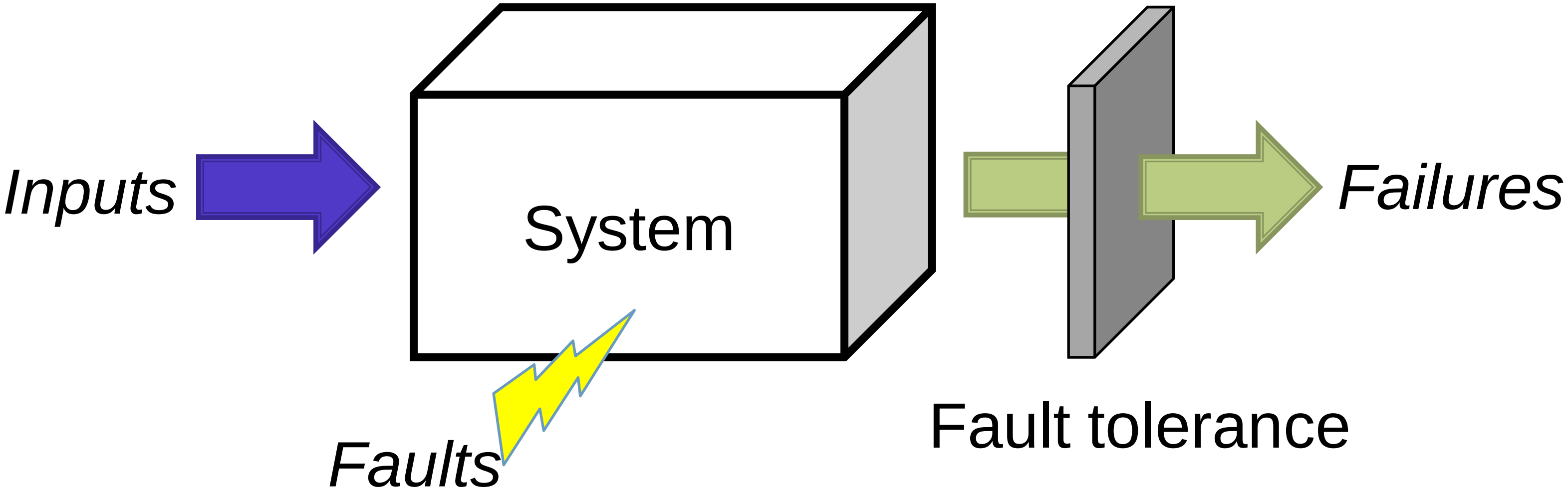
Software Fault Tolerance

- **Error detection and handling mechanisms** cope with residual defects by:
  1. Masking software faults
     - N-version programming, recovery blocks, ...
  2. Detecting an incorrect state, in order to provide a **fail-stop behavior** or a **degraded mode of service**
     - Assertions, watchdog timers, time and space partitioning, exception handling, ...

- They also require **testing and debugging**, and **evidences** proving their effectiveness

Fault injection

- **Fault Injection** is the process of **deliberately introducing faults** into a system to assess its behavior in the presence of faults



Fault injection in the ISO/DIS 26262 safety standard

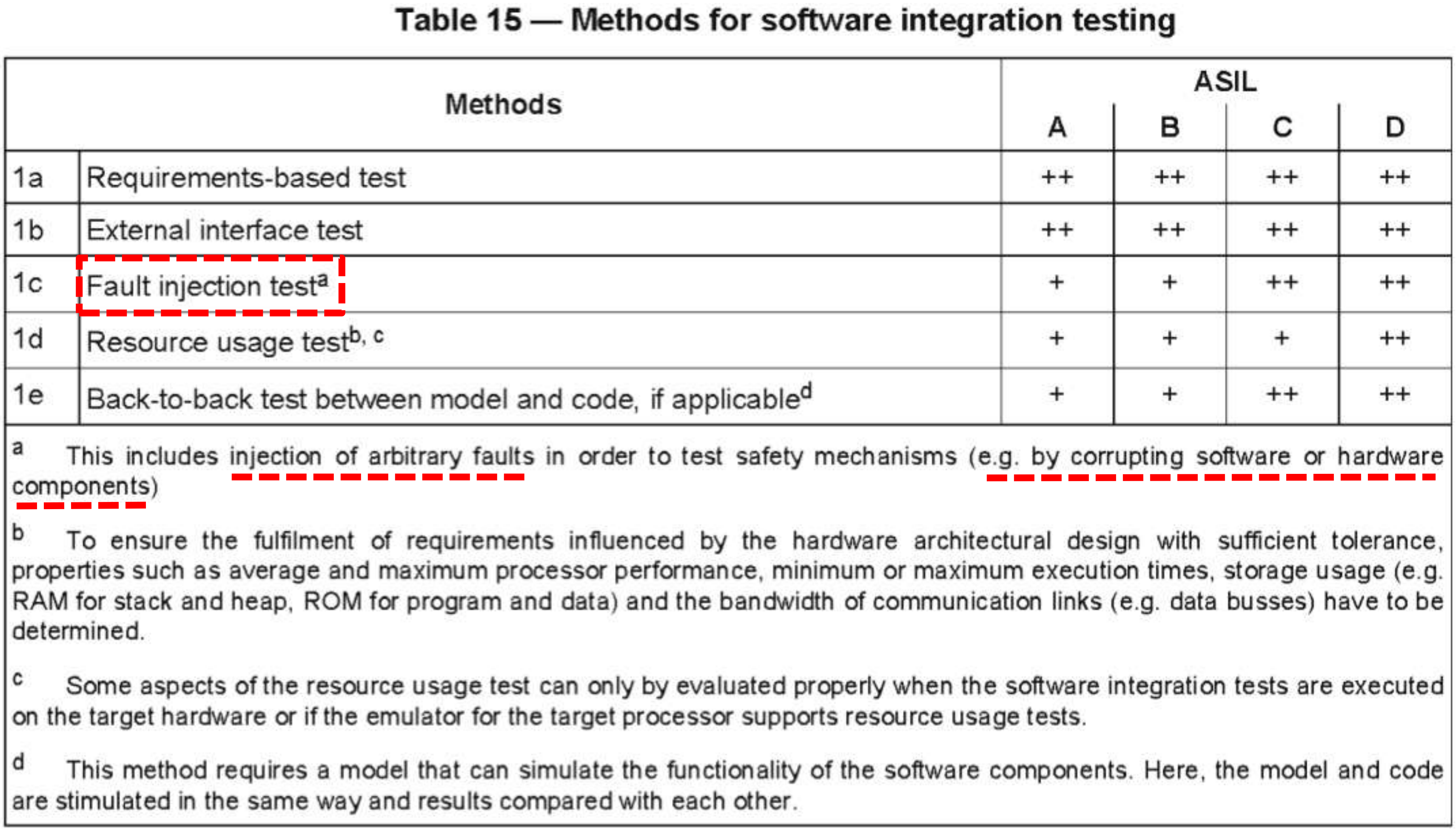


Table 15 — Methods for software integration testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test	++	++	++	++
1b	External interface test	++	++	++	++
1c	Fault injection test[a]	+	+	++	++
1d	Resource usage test[b, c]	+	+	+	++
1e	Back-to-back test between model and code, if applicable[d]	+	+	++	++

[a] This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software or hardware components)

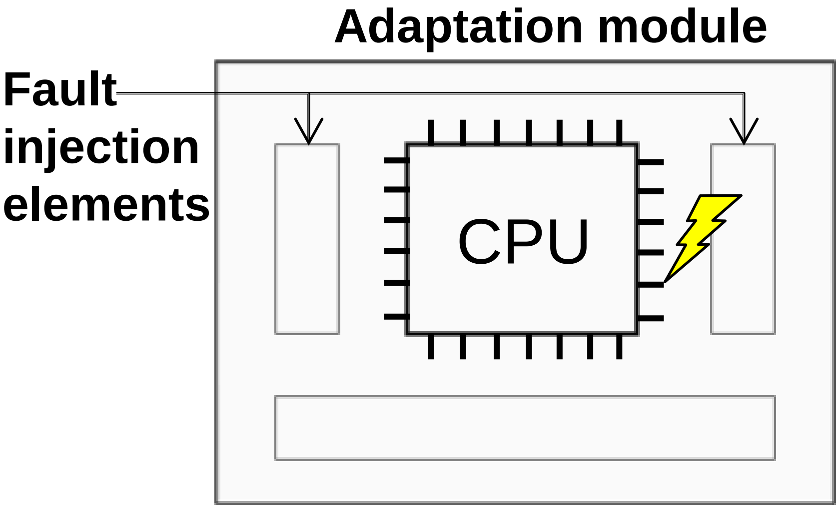
[b] To ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for program and data) and the bandwidth of communication links (e.g. data busses) have to be determined.

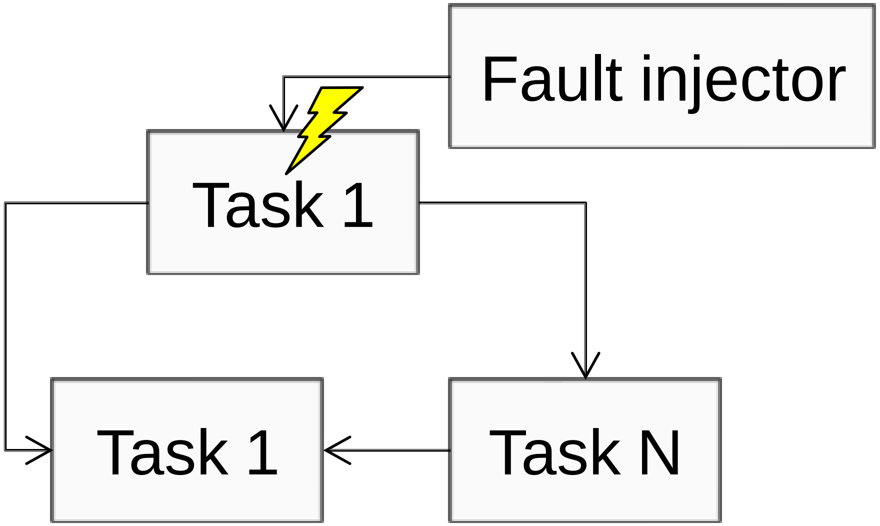
[c] Some aspects of the resource usage test can only by evaluated properly when the software integration tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.

Source: ISO/DIS 26262-6, “*Product Development: software level*”, 2009 (draft version)

Traditional hardware fault injection





Hardware-implemented fault injection
(e.g., pin-level injection)

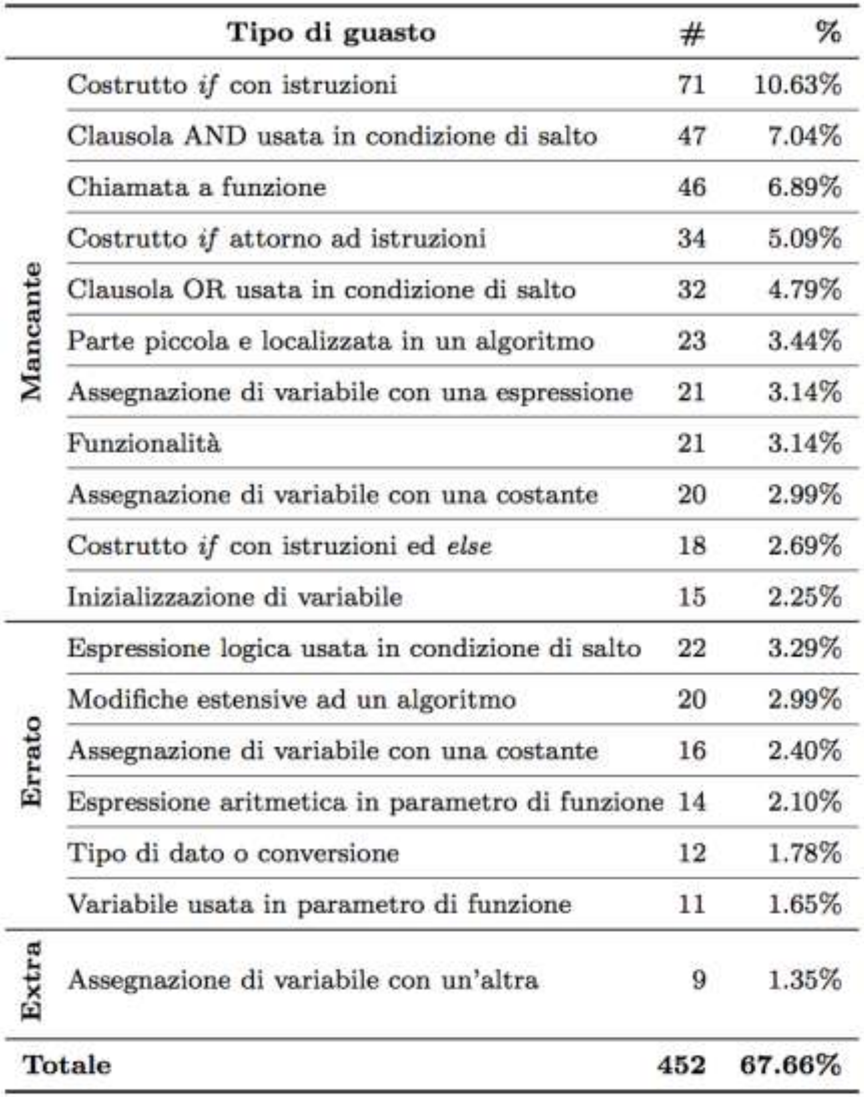
Software-implemented fault injection
(e.g., bit-flipping)

Injection of software faults

- Software faults are more complex to emulate than hardware faults
- They are human mistakes occurring in the **development process**

```
static void tg3_read_mem(struct tg3 *tp, u32 off, u32 *val) {
        unsigned long flags;

        if ((GET_ASIC_REV(tp->pci_chip_rev_id) == ASIC_REV_5906) &&
            (off >= NIC_SRAM_STATS_BLK) && (off < NIC_SRAM_TX_BUFFER_DESC)) {
                *val = 0;
                return;
        }

        spin_lock_irqsave(&tp->indirect_lock, flags);
        if (tp->tg3_flags & TG3_FLAG_SRAM_USE_CONFIG) {
                pci_write_config_dword(tp->pdev, TG3PCI_MEM_WIN_BASE_ADDR, off);
                pci_read_config_dword(tp->pdev, TG3PCI_MEM_WIN_DATA, val);

                /* Always leave this as zero. */
                pci_write_config_dword(tp->pdev, TG3PCI_MEM_WIN_BASE_ADDR, 0);
        } else {
                tw32_f(TG3PCI_MEM_WIN_BASE_ADDR, off);
                *val = tr32(TG3PCI_MEM_WIN_DATA);

                /* Always leave this as zero. */
                tw32_f(TG3PCI_MEM_WIN_BASE_ADDR, 0);
        }
        spin_unlock_irqrestore(&tp->indirect_lock, flags);
}
```

Characterization of software faults

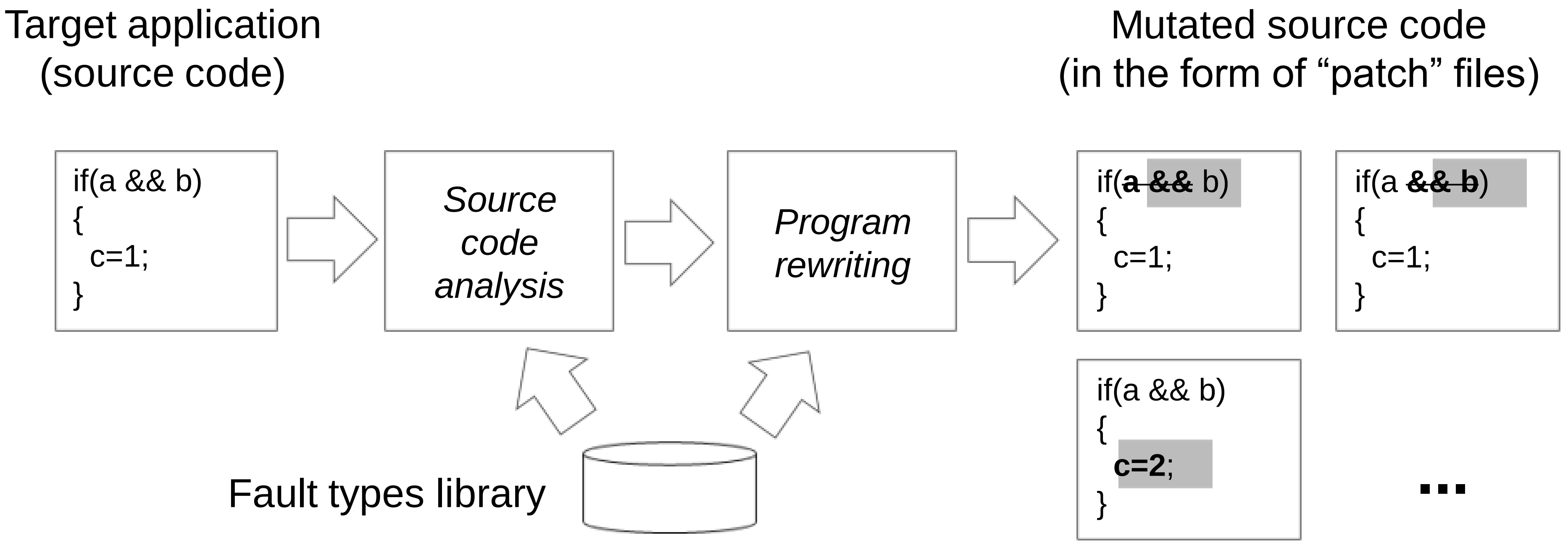


	Tipo di guasto	#	%
Mancante	Costrutto *if* con istruzioni	71	10.63%
	Clausola AND usata in condizione di salto	47	7.04%
	Chiamata a funzione	46	6.89%
	Costrutto *if* attorno ad istruzioni	34	5.09%
	Clausola OR usata in condizione di salto	32	4.79%
	Parte piccola e localizzata in un algoritmo	23	3.44%
	Assegnazione di variabile con una espressione	21	3.14%
	Funzionalità	21	3.14%
	Assegnazione di variabile con una costante	20	2.99%
	Costrutto *if* con istruzioni ed *else*	18	2.69%
	Inizializzazione di variabile	15	2.25%
Errato	Espressione logica usata in condizione di salto	22	3.29%
	Modifiche estensive ad un algoritmo	20	2.99%
	Assegnazione di variabile con una costante	16	2.40%
	Espressione aritmetica in parametro di funzione	14	2.10%
	Tipo di dato o conversione	12	1.78%
	Variabile usata in parametro di funzione	11	1.65%
Extra	Assegnazione di variabile con un'altra	9	1.35%
Totale		**452**	**67.66%**

- A large set of bugs in commercial and open-source software was used to characterize software faults
- Faults were classified as **missing**, **wrong**, or **extraneous** constructs
- The majority of faults (68%) belongs to a set of few **fault types**

SoftwAre Fault Emulation (SAFE)

CRITIWARE s.r.l.
Innovating Critical Systems Engineering



Target application
(source code)

if(a && b)
{
c=1;
}

Source code analysis

Program rewriting

Fault types library

Mutated source code
(in the form of “patch” files)

if(~~**a &&**~~ b)
{
c=1;
}

if(a ~~**&& b**~~)
{
c=1;
}

if(a && b)
{
c=2;
}

...

- An industrial-strength C/C++ parser (tested on the Linux kernel, MySQL, Apache, …) automatically analyzes the source code, to identify “injectable” code locations
- “Patch files” are automatically generated, each introducing an individual fault

Workflow

```
$ ./injection main.c
$ ls
injection              main.ii_OMIA_0.patch   main.ii_OMVAE_0.patch  main.ii_OWPFV_1.patch  test.h
main.c                 main.ii_OMIFS_0.patch  main.ii_OMVIV_0.patch  main.o                 test.ii
main.ii                main.ii_OMLAC_0.patch  main.ii_OMVIV_1.patch  main.s                 test.o
main.ii_OMFC_0.patch   main.ii_OMLAC_1.patch  main.ii_OMVIV_2.patch  test                   test.s
main.ii_OMFC_1.patch   main.ii_OMLPA_0.patch  main.ii_OWPFV_0.patch  test.c
```

1. Several “patch” files are generated

```
$ cat main.ii_OMVAE_0.patch
--- /home/pippo/Scrivania/test/main.c
+++ /home/pippo/Scrivania/test/main.c
@@ -12,1 +12,1 @@
-    punt = &a;
+    punt =  (punt);

$ patch -p0 < main.ii_OMVAE_0.patch
patching file /home/pippo/Scrivania/test/main.c
```

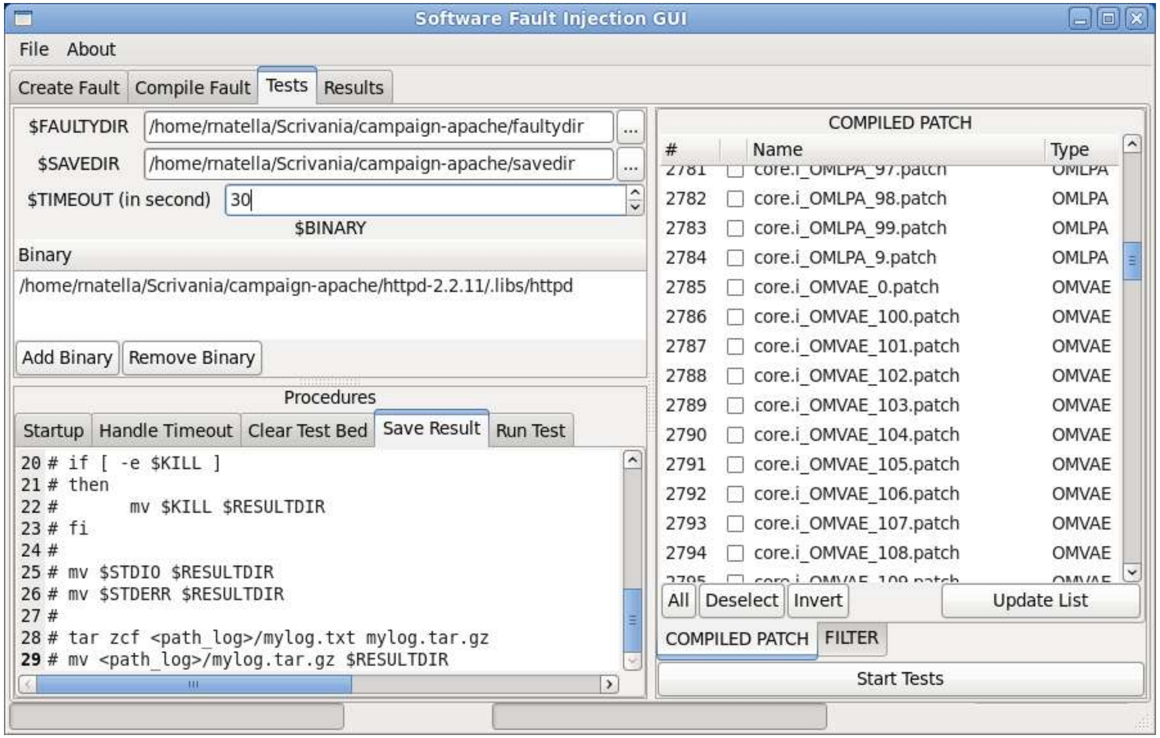
2 A “patch” is applied to the software

```
$ make
$ ./test
Segmentation fault (core dumped)
```

3. Test execution

Automation of Fault Injection Tests

CRITIWARE s.r.l.
Innovating Critical Systems Engineering



$FAULTYDIR: /home/rnatella/Scrivania/campaign-apache/faultydir

$SAVEDIR: /home/rnatella/Scrivania/campaign-apache/savedir

$TIMEOUT (in second): 30

$BINARY

Binary: /home/rnatella/Scrivania/campaign-apache/httpd-2.2.11/.libs/httpd

Procedures

```
# if [ -e $KILL ]
# then
#       mv $KILL $RESULTDIR
# fi
#
# mv $STDIO $RESULTDIR
# mv $STDERR $RESULTDIR
#
# tar zcf <path_log>/mylog.txt mylog.tar.gz
# mv <path_log>/mylog.tar.gz $RESULTDIR
```

COMPILED PATCH

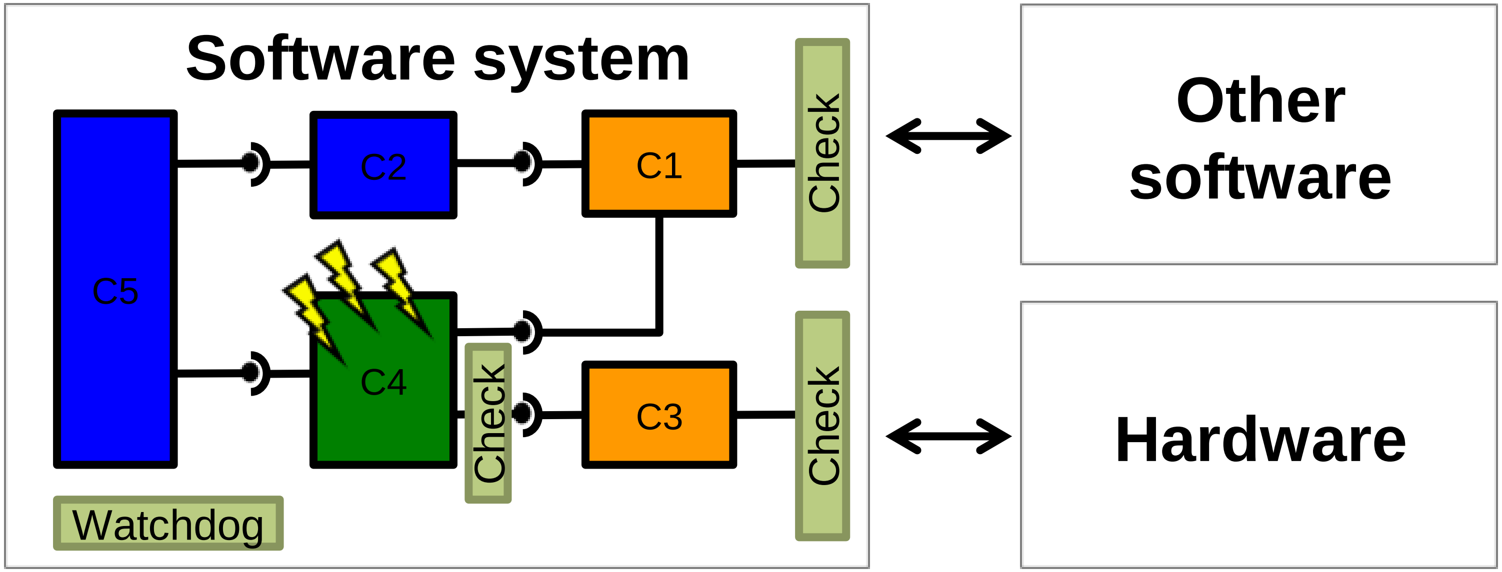
#		Name	Type
2781	☐	core.i_OMLPA_97.patch	OMLPA
2782	☐	core.i_OMLPA_98.patch	OMLPA
2783	☐	core.i_OMLPA_99.patch	OMLPA
2784	☐	core.i_OMLPA_9.patch	OMLPA
2785	☐	core.i_OMVAE_0.patch	OMVAE
2786	☐	core.i_OMVAE_100.patch	OMVAE
2787	☐	core.i_OMVAE_101.patch	OMVAE
2788	☐	core.i_OMVAE_102.patch	OMVAE
2789	☐	core.i_OMVAE_103.patch	OMVAE
2790	☐	core.i_OMVAE_104.patch	OMVAE
2791	☐	core.i_OMVAE_105.patch	OMVAE
2792	☐	core.i_OMVAE_106.patch	OMVAE
2793	☐	core.i_OMVAE_107.patch	OMVAE
2794	☐	core.i_OMVAE_108.patch	OMVAE

- A huge number of tests can be automatically performed in few days

	Size (KLoC)	# faults	Time/test
MySQL	232	39,539	~3 sec.
PostgreSQL	367	32,915	~10 sec.
Apache	26	11,621	~11 sec.

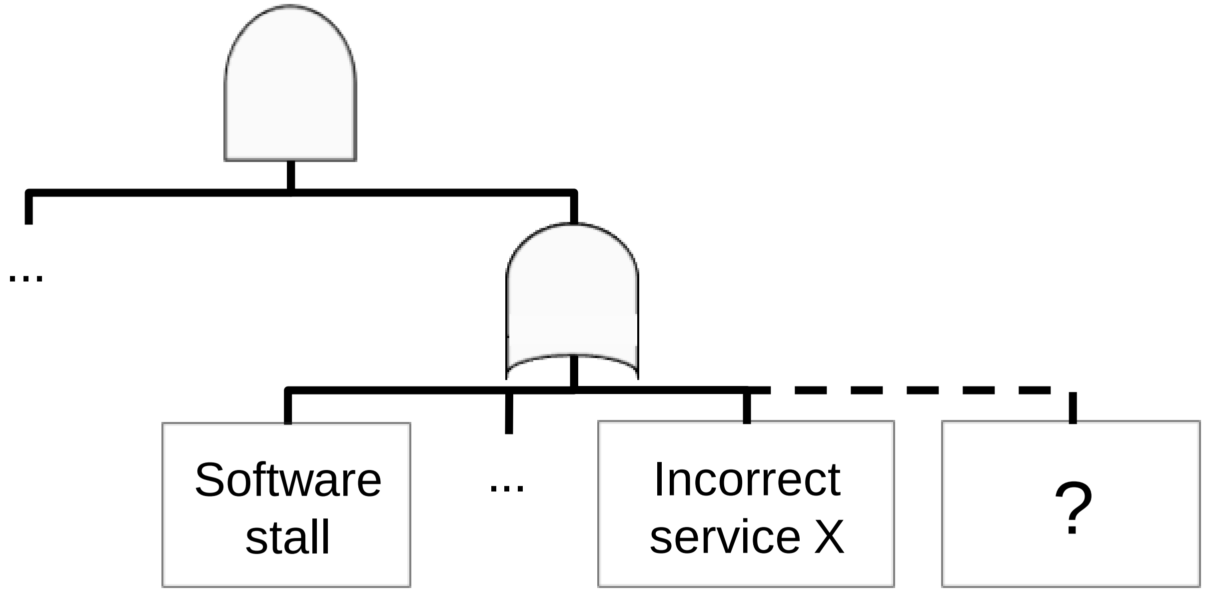
Applications in Software Certification (1/2)

- Verification&Validation of Software Fault Tolerance mechanisms and algorithms
  - Testing and debugging
  - Evidence of their effectiveness



Applications in Software Certification (2/2)

- Validation of failure mode analysis (e.g., FMECA, Fault Trees)
  - Software failure modes are **not completely known** and difficult to identify, and they depend on the specific software component
  - Need to provide evidence that all likely failure modes have been covered (e.g., by **emulating real defects in software components**)

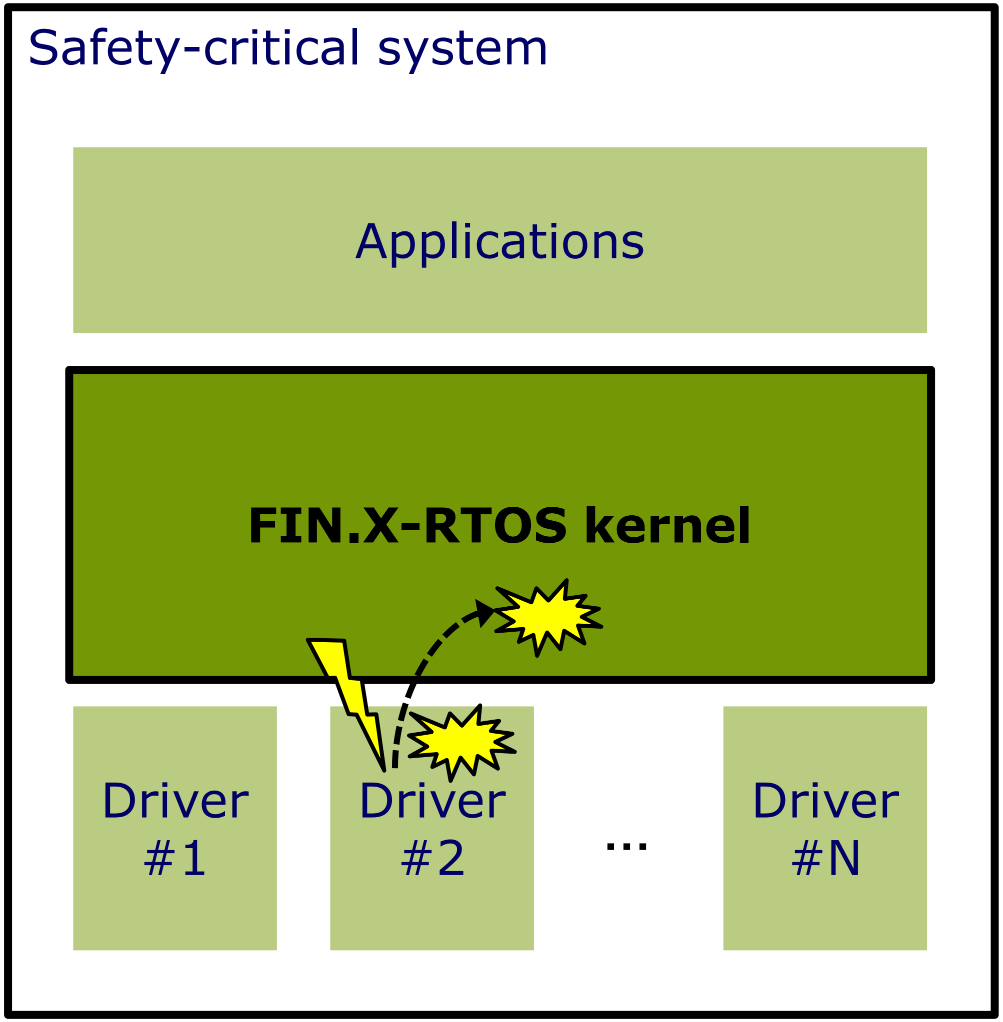


Case study

CRITIWARE s.r.l.
Innovating Critical Systems Engineering

- **FIN.X-RTOS** is a real-time OS based on the Linux kernel from Finmeccanica
- Aim of this project is to provide an OS compliant with the guidelines of the **DO-178B safety standard**
  - Safety evidences will be used for certifying systems based on FIN.X-RTOS
  - Level D requirements already fulfilled, level C is being considered

OS robustness against faulty drivers



- Device drivers:
  - are **bug-prone** components (3 to 7 times buggier than other components)
  - run in **supervisor** mode
  - are tightly coupled through **APIs** and **shared data**
- Software Fault Injection adopted for evaluating if faults can spread to the kernel
  - Propagation to other kernel components
  - Silent kernel data corruption

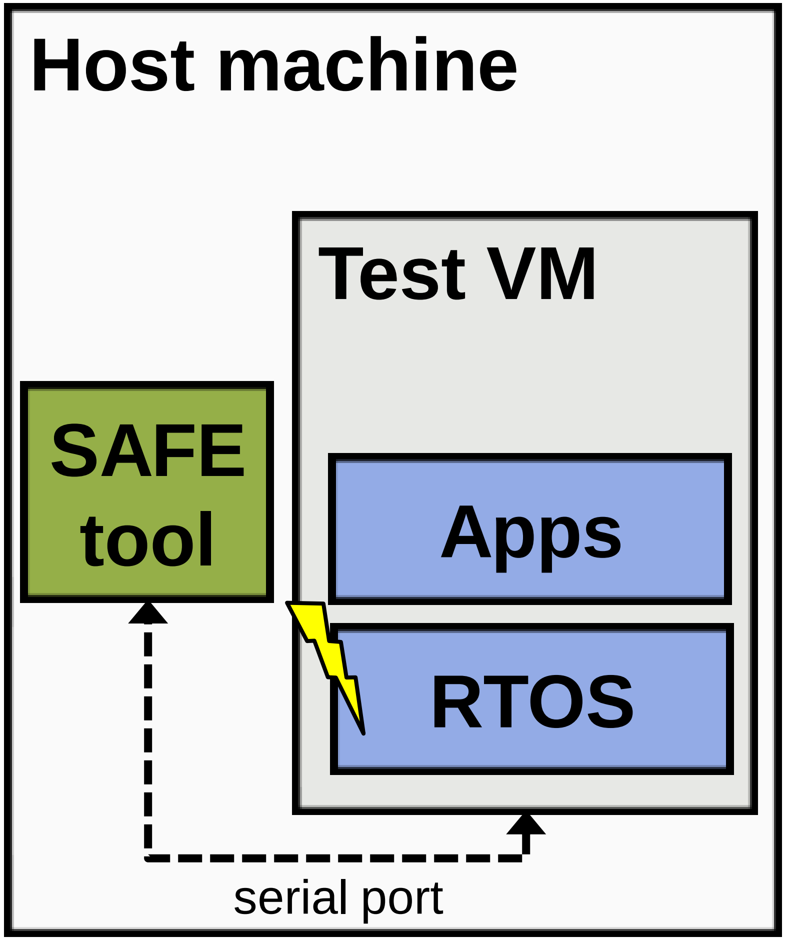
Test campaign

CRITIWARE s.r.l.
Innovating Critical Systems Engineering

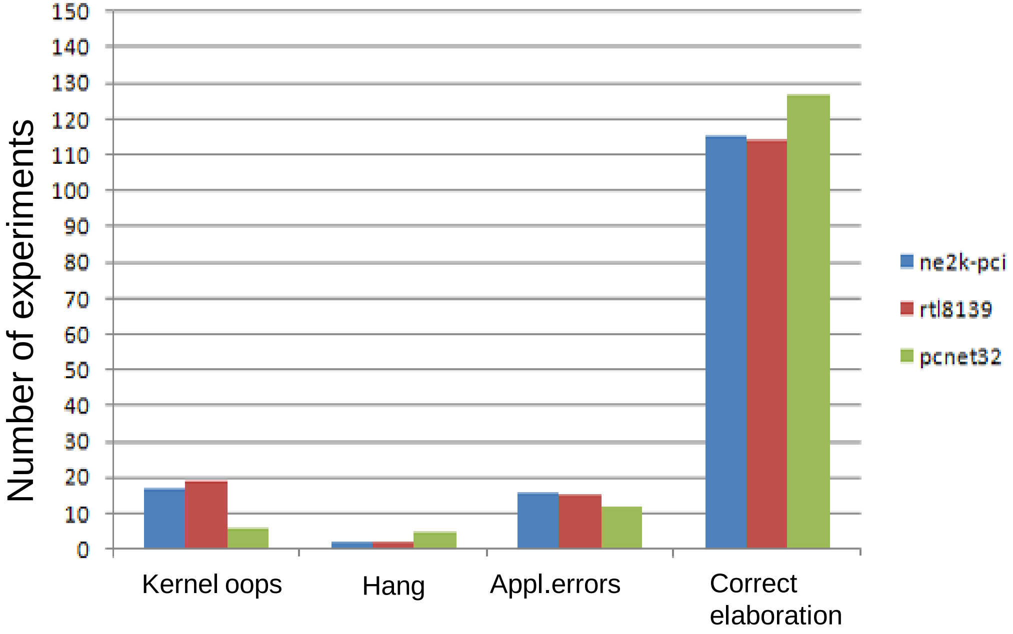
a. For each injectable fault:
   1. Generation of a "faulty driver" by injecting the fault in the original driver
   2. Installation and loading of the driver in the kernel
   3. Execution of an user application
   4. Data collection (error messages from kernel/apps; register and memory dumps)

b. Analysis of kernel failure modes

- Fault injection in 3 **network device drivers** (ne2k-pci, rtl8139cp, pcnet32)
- 150 injected faults per device driver



Test results (1/2)



- Classification of failure modes:
  - Kernel oops
  - Hang (stall)
  - Application errors
- More than half of the failures impact on the kernel state (kernel oops and hangs)

Test results (2/2)

- Analysis of kernel error messages and register/memory dumps:
  - 46/51 error messages denote a failure **within the device driver**
    - *These failures can be tolerated by unloading the driver, releasing its resources (locks, memory), and reloading the driver*
  - 5/51 error messages denote a failure in **other kernel components**
    - *Errors propagated to the rest of the kernel; more checks may be needed in kernel primitives involved in these failures*

Concluding remarks

- **Residual faults** are hidden in our software, and **they will eventually manifest themselves during operation**
- Software Fault Injection is a means to **assess and mitigate their impact before releasing the product**
- It is a reasonably mature technology that can be adopted in complex software systems

Thank you for the attention!

Roberto Natella

roberto.natella@critiware.com

CRITIWARE s.r.l.
Innovating Critical Systems Engineering