HW Fault Injection Mitigation

Trusted Firmware M

Tamas Ban
Arm

Agenda

- Fault Injection overview
- Software countermeasures
- MCUBoot overwiew
- SW countermeasures in MCUBoot
- QEMU based test tool

A high-level view on fault injection

A fault is physical perturbation altering the correct / expected behaviour of a circuit.

It can be a change in voltage or temperature, or a laser beam, or an EM pulse,... All have different effects.

Effect can be permanent (damage) or transient

Physical access is **not** always needed

- rowhammer or clkscrew for example

Strongly correlated with reliability:

- Reliability is about “random” hazards
- Fault injection is about an adversary actively introducing hazards

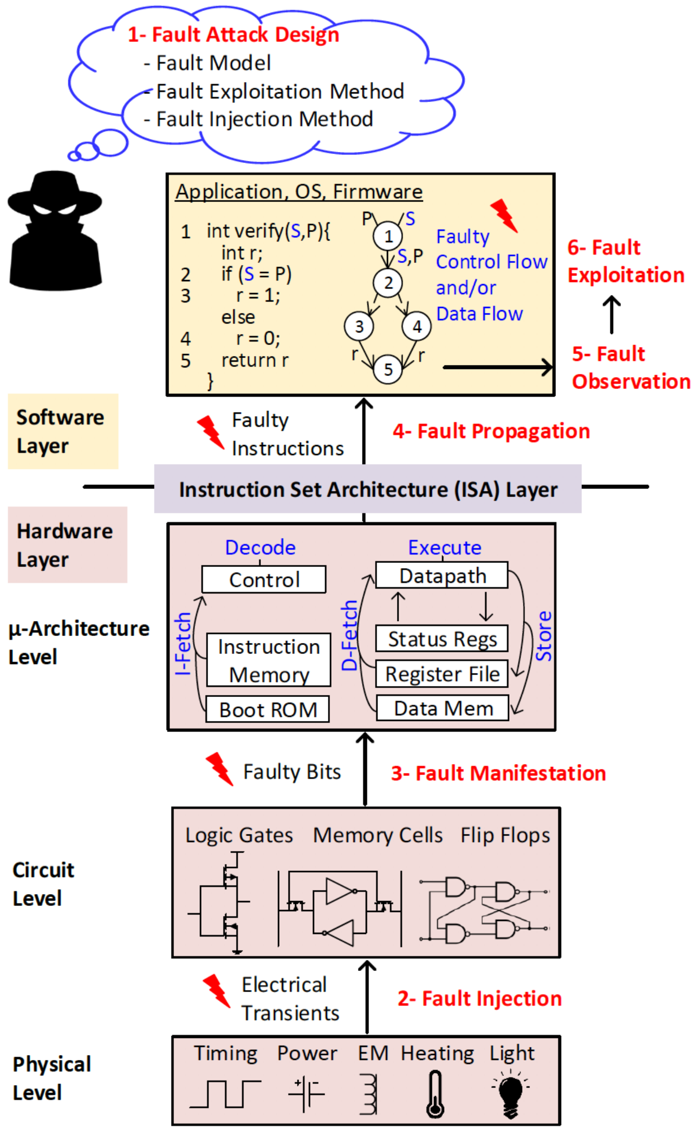


Figure from “Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation”, *Bilgiday Yuce, Patrick Schaumont, Marc Witteman*

Slide from Arnaud De Grandmaison

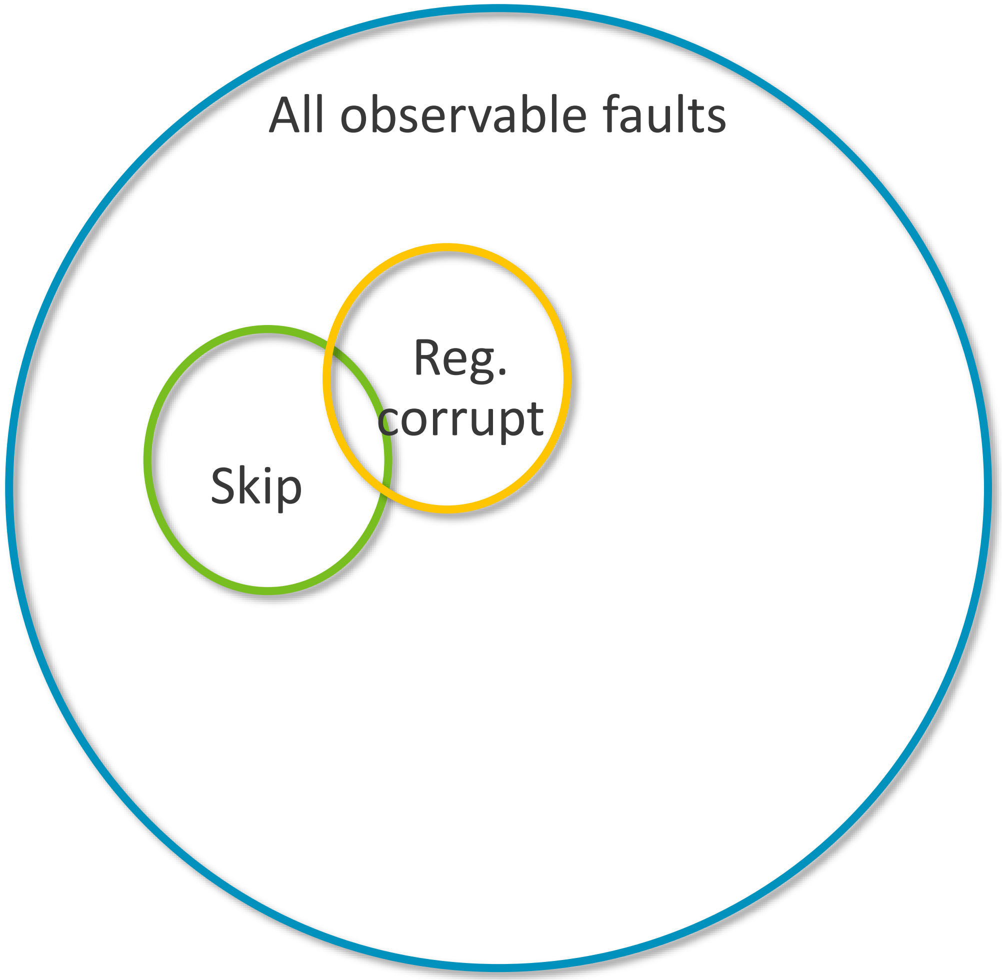
A high-level view on fault injection (cont.)

This is a complex domain!

- Faults are not well understood
- This is an active (but niche) research domain

All models are wrong --- but each one address a specific aspect of some observed faults and is thus useful

Ultimately it's all about using different models to explore and reason about the unknown / complex



Slide from Arnaud De Grandmaison

Software countermeasures

- The objective is to **protect against unautheticated code execution**.
- There are **dedicated hardware** components which can provide a level of protections, but there is an additional level of **defense provided by software** countermeasures – **defense-in-depth approach**.
- Although **there is no way guarantee defense** from those attacks neither by hardware nor by software, the more countermeasure there are in place, the harder are attacks.
- There are practical techniques that can be applied to the coding and **significantly decrease the probability of successful attacks**.

Generic countermeasures

- Side channel attacks
  - Timing information leakage prevention
  - Secrets leakage prevention
- **Fault injection attacks**
  - **Complex (large hamming distance) constants**: More bit need to be flip to change one valid value to another.
  - **Double checks, switch/case double checks**: Make harder to attack the branch conditions, check same condition twice.
  - **Loop integrity checks**: Make sure important loops are executed, check expected index value after the loop.
  - **Default failure**: Skipping instructions or attacking PC can bypass important code. Default return value is failure.
  - **Flow monitor**: Global counter is incremented and its expected value checked to make sure that expected flow is executed.
- Good resources in the topic:
  - https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf
  - https://www.riscure.com/uploads/2018/11/201708_Riscure_Whitepaper_Side_Channel_Patterns.pdf

How to do fault injection in pratice?

- Albeit FI seems a mystery, many-many resources available how to perform it.
- Even commercial tools are available to break devices with FI.
- SW framework with scripting support to automate attack execution.
- Tutorials

Is there a SW lib to harden my code?

- Generic solution does not exist.
- Compilers makes it impossible.
- Compiled code depends on HW architecture, actual compiler, optimization level, etc.
- **Compiled code must be verified.** On C level seems safe, but the binary might not...

Why MCUBoot is hardened primarily?

- TF-M consist of (roughly):
  - Secure boot code: MCUboot
  - Runtime SW: Secure partiton manager & Secure partitions
- Secure boot code has a time deterministic execution. With physical access easy to try 1000x time to break the device.
- With right timing the image authentication can be bypassed and all secrets could be disclosed from the device.
- Vulnerable function calls in the boot flow.

```
rc = boot_go(&rsp);
if (rc != 0) {
    BOOT_LOG_ERR("Unable to find
                  bootable image");
    while (1)
        ;
}
do_boot();
```

Reset register

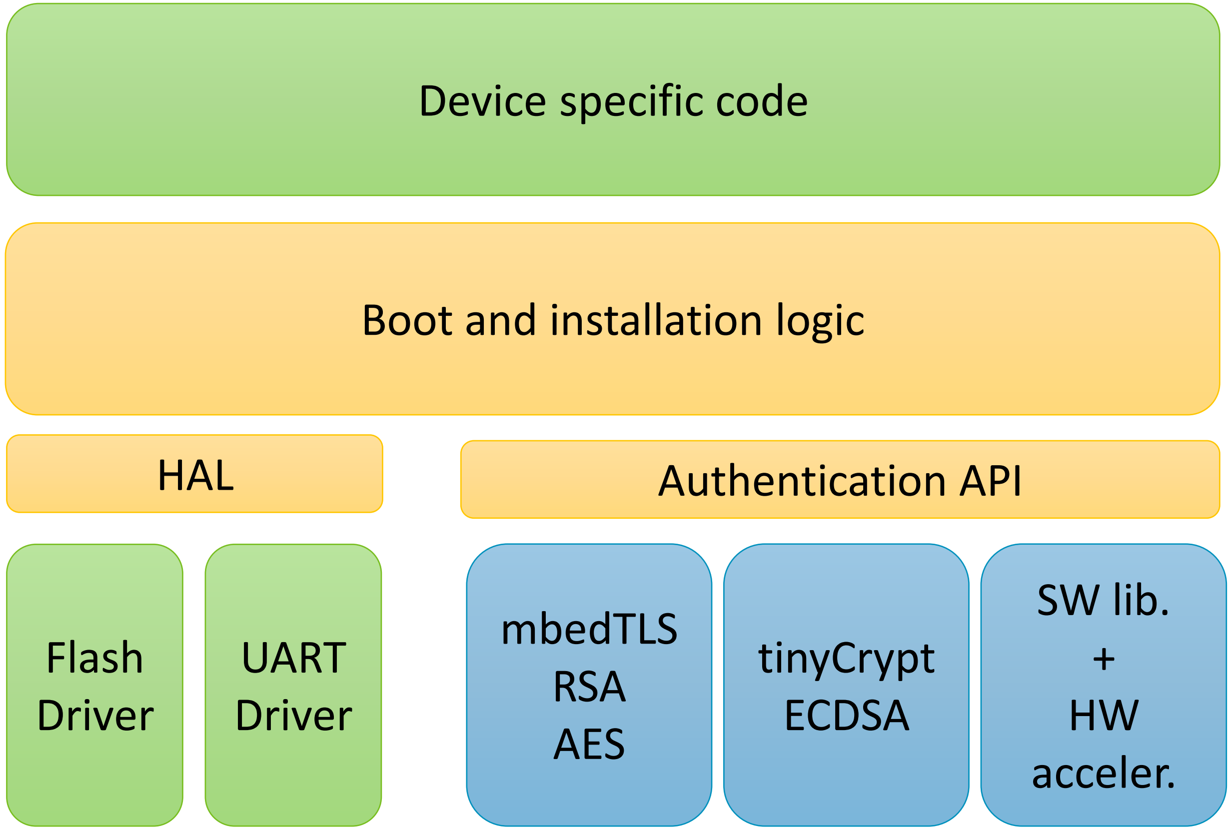
Skip instructions

Reset zero flag in status reg.

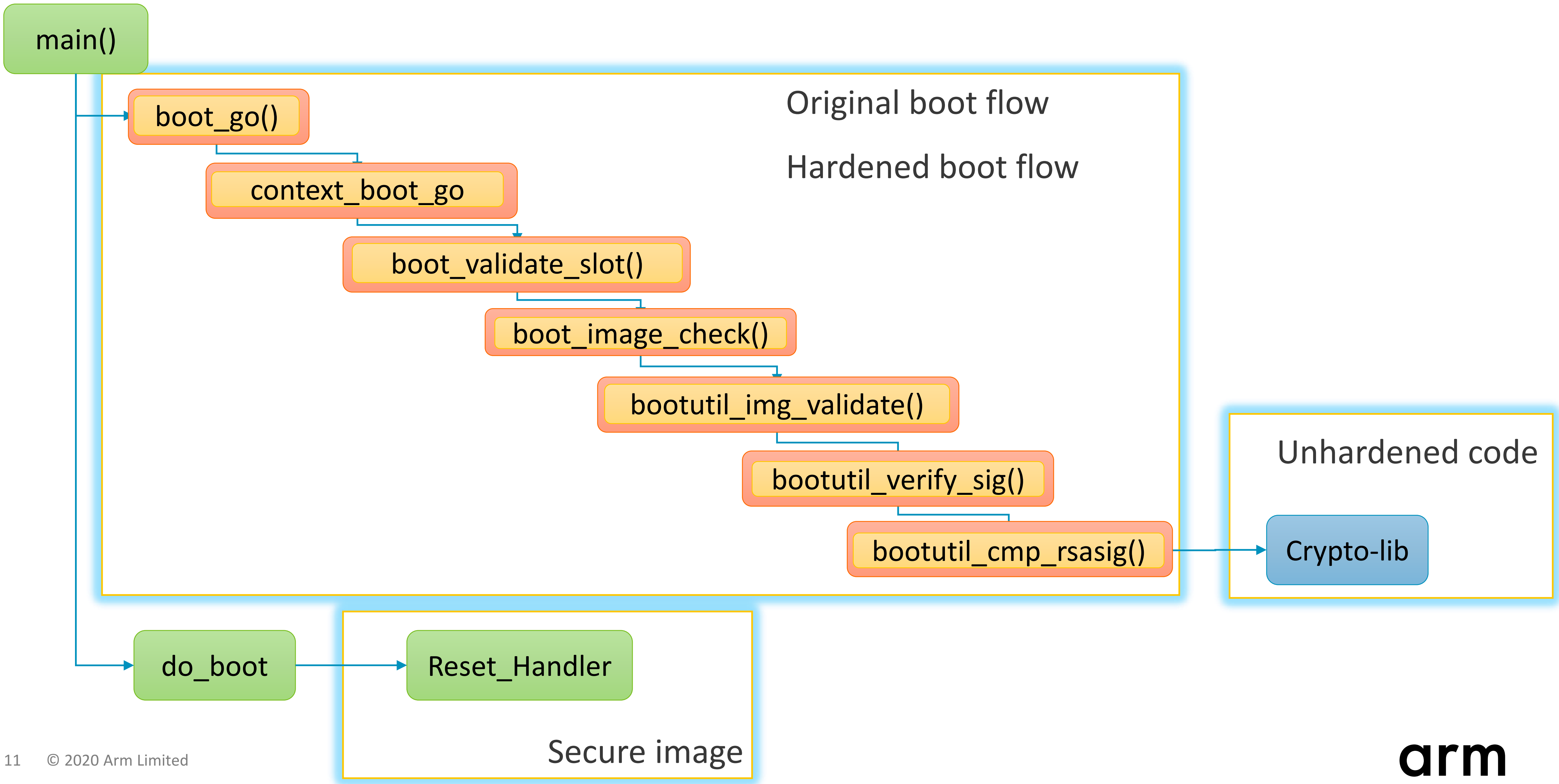
Jump out from error loop with instruction skip

MCUBoot overview

- Designed to 32bit MCUs
- Low memory footprint (~18KB of ROM)
- Compatible with several crypto library (mbedTLS, tinyCrypt)
- RSA, ECDSA support
- Encrypted image support
- Custom image manifest format (TLV)
- No X.509 support, No SUIT manifest support
- No fault injection or side channel attack protection so far



Boot flow


main()
boot_go()
context_boot_go
boot_validate_slot()
boot_image_check()
bootutil_img_validate()
bootutil_verify_sig()
bootutil_cmp_rsasig()
Original boot flow
Hardened boot flow
Unhardened code
Crypto-lib
do_boot
Reset_Handler
Secure image

Where we are?

- Beginning of learning process
- Added hardening to MCUboot generic code(bootutil). Configurable at 4 level:
  - https://github.com/JuulLabs-OSS/mcuboot/pull/776
- Have a QEMU based fault injection test tool (only instruction skip fault model):
  - https://github.com/JuulLabs-OSS/mcuboot/pull/789
- With SW hardening the boot process is more secure (MCUboot + TF-M Release build):

	Image size	Executed tests	Boots with corrupted image
MCUBOOT_FIH_PROFILE_OFF	Flash: 25.1 kB RAM: 25.4 kB	560	31 (5.5%)
MCUBOOT_FIH_PROFILE_LOW	Flash: 25.5 kB RAM: 25.4 kB	855	12 (1.4%)
MCUBOOT_FIH_PROFILE_MEDIUM	Flash: 27.7 kB RAM: 25.4 kB	1275	3 (0.2%)

SW countermeasures in MCUBoot

- Primitives added to harden existing code
- Only added to critical code path
- Build time configurable, 4 profiles available(HIGH, MEDIUM, LOW, OFF)

Countermeasure	Status	Profile
Control flow integrity	Implemented	LOW
Failure loop hardening	Implemented	LOW
Complex constants	Implemented	MEDIUM
Redundant variables and checks	Implemented	MEDIUM
Random delay	Implemented, but depends on device capability.	HIGH
Loop integrity checks	Not implemented	-

arm

Countermeasures are C code

- People in the real world don't like security when it gets in the way
- Have to support three compilers and both armv8m and armv6m
- All protections implemented in two macros and one typedef
- Code size increase with all countermeasures disabled only 250 bytes
- Verified asm under GCC and ARMCLANG although this may break with future versions
- Much better than nothing

Critical call path hardening

```c
rc = boot_go(&rsp);
if (rc != 0) {
    BOOT_LOG_ERR("Unable to find
                  bootable image");
    while (1)
        ;
}
```

```c
FIH_CALL(boot_go, fih_rc, &rsp);
if (fih_not_eq(fih_rc, FIH_SUCCESS)) {
    BOOT_LOG_ERR("Unable to find
                  bootable image");
    FIH_PANIC;
}
```

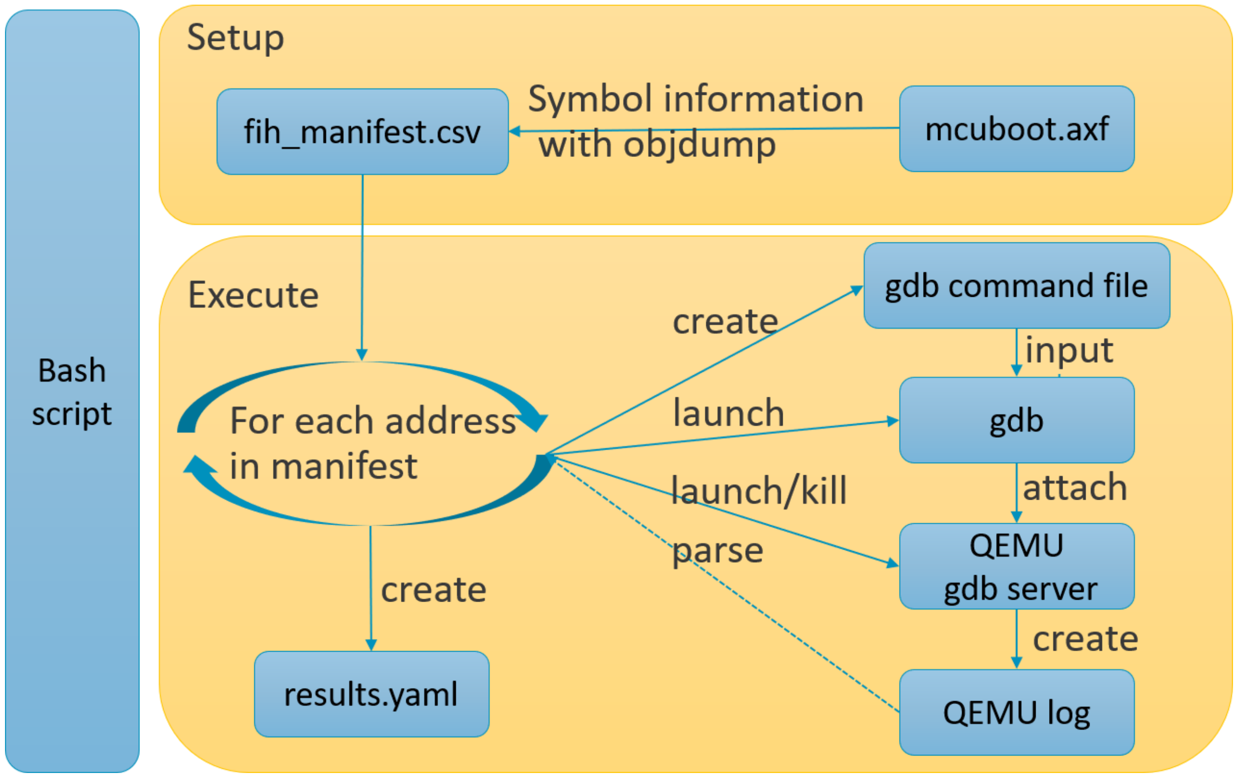
```c
#define FIH_CALL(f, ret, ...) \
    do { \
        FIH_LABEL("START"); \
        FIH_CFI_PRECALL_BLOCK; \
        ret = FIH_FAILURE; \
        if (fih_delay()) { \
            ret = f(__VA_ARGS__); \
        } \
        FIH_CFI_POSTCALL_BLOCK; \
        FIH_LABEL("END"); \
    } while (0)
```

```c
#define FIH_RET(ret) \
    do { \
        FIH_CFI_PRERET; \
        return ret; \
    } while (0)
```


arm

QEMU based fault injection test tool

- Easy integration with CI, faster and reliable than HW, different builds (opt levels) and compilers can be tested in short time.
- Code is annotated with labels to indicate where to test.
- Labels are part of the hardening code, they are included automatically.
- START / END labels are extracted to get addresses to test in that range.
- Bash script launches QEMU and interacts with it over gdb
- Test tries to boot a tampered image
- Instruction skip fault model as this is the most common and cheapest attack to perform
- Serial output is parsed and evaulated



Potential enhancements

- Implements new fault models: Resetting registers at certain pattern (CMP r0, #0)
- Expand testing beyond START/END labels to increase coverage:
  - i.e: List of potentialy voulnarebel files/functions.
- Implement testing on HW.

arm