2017 Workshop on Fault Diagnosis and Tolerance in Cryptography

# Escalating Privileges in Linux using Voltage Fault Injection

Niek Timmers
*Riscure – Security Lab*
*timmers@riscure.com / @tieknimmers*

Cristofaro Mune
*Embedded Security Consultant*
*pulsoid@icysilence.org / @pulsoid*

***Abstract*—Today's standard embedded device technology is not robust against Fault Injection (FI) attacks such as Voltage Fault Injection (V-FI). FI attacks can be used to alter the intended behavior of software and hardware of embedded devices. Most FI research focuses on breaking the implementation of cryptographic algorithms. However, this paper's contribution is in showing that FI attacks are effective at altering the intended behavior of large and complex code bases like the *Linux Operating System (OS)* when executed by a fast and feature rich System-on-Chip (SoC). More specifically, we show three attacks where full control of the *Linux OS* is achieved from an unprivileged context using V-FI. These attacks target standard *Linux OS* functionality and operate in absence of any logical vulnerability. We assume an attacker that already achieved unprivileged code execution. The practicality of the attacks is demonstrated using a commercially available V-FI test bench and a commercially available ARM Cortex-A9 SoC development board. Finally, we discuss mitigations to lower probability and minimize impact of a successful FI attack on complex systems like the *Linux OS*.**

***Keywords*-Fault Injection; ARM; Linux.**

## I. Introduction

Embedded devices are commonly found to be vulnerable to logical attacks due to exploitable vulnerabilities in software. Multiple initiatives exist (e.g. *Coordinated Disclosure* [1] and *Bug Bounty Programs* [2]) where security researchers are invited to disclose such vulnerabilities, allowing them to be fixed by the manufacturer. Assuming such initiatives are effective, they should result, in the long term, in software without exploitable vulnerabilities. Even though the absence of exploitable vulnerabilities in today's embedded technology is unlikely, it is interesting to research attack techniques that do not rely on their presence. An example of such an attack technique is FI.

Today's embedded technology is typically designed around a SoC that includes a fast processing core to execute software. Common architectures for implementation are ARM and MIPS. The architectures are produced in a multitude of flavors including 32 bits and 64 bits variants to accommodate today's processing power needs. Altering the intended behavior of code executed within the *Linux OS* using V-FI is not new. In [3] the authors show it is possible to break several cryptographic algorithms executed within the *Linux OS* by an older generation ARM9 processor. This paper's contribution is showing that V-FI is an effective attack technique for escalating privileges within the *Linux OS* by modifying its intended behavior using *V-FI*. An attacker with physical access to the target can use this attack technique to compromise the *Linux OS* from an unprivileged context.

Like many OSes, the (standard) *Linux OS* [4] has two execution modes: *User Mode* and *Kernel Mode*. The *Linux Kernel* is executed in *Kernel Mode* and has no restrictions for accessing hardware or memory. All Linux applications are executed in *User Mode* and can only access hardware and memory indirectly by leveraging *Linux Kernel* functionality. The *Linux Kernel* prevents unprivileged Linux applications to access functionality that may lead to a compromised *Linux OS*. These restrictions are often implemented using a single code construction which makes them interesting for single glitch FI attacks.

All attacks are demonstrated using a commercially available development board, from now on referred to as *Target*, which is designed around a fast and feature rich ARM Cortex-A9 processor SoC. A commercially available V-FI test bench is used to perform *V-FI* on the *Target*. The processor operates at 1 GHz and the *instruction cache* and *data cache* are by default enabled. All attacks described in this paper are executed from external DDR3 unless cached. After power-on reset the processor executes the following boot stages: a Read Only Memory (ROM) loader, *U-Boot 2017.01-00047*, *Linux Kernel 4.9.16-armv7-x4* and *Ubuntu 16.04.02 32-bit*. The operating system is updated using *apt-get update && apt-get upgrade*.

This paper introduces FI concepts. An initial *FI* characterization phase shows that the *Target* is susceptible to V-FI. During this characterization we target code executed under different conditions. Then, three attacks are identified and demonstrated, where standard *Linux OS* functionality is targeted to escalate privileges from an unprivileged to a privileged context. All the three attacks potentially result in full *Target* control. To finalize, we discuss mitigations for lowering the probability and minimizing the impact of a successful FI attack on larger code bases like the *Linux OS*.

## II. Fault Injection Background

FI attacks alter a target's intended behavior by manipulating its environmental conditions. This can be accomplished by using different techniques. For example, common techniques include: *Clock FI* [5] [6], *V-FI* [7] [8],

978-1-5386-2948-2 /17 $31.00 © 2017 IEEE
DOI 10.1109/FDTC.2017.16

*Electromagnetic FI* [9] [10] and *Optical FI* [11] [12]. This paper primarily focuses on V-FI. However, the results hereby described may be applicable for other FI techniques as well. The manipulation of the target's environmental conditions is from now on referred to as the *glitch*. The behavioral alteration in the target is from now on referred to as the *fault*.

A target's behavior can be altered in different ways using FI. The different faults can be classified into two groups: *faults affecting hardware able to execute software* and *faults affecting hardware that does not execute software*. Targeting a subsystem responsible for software execution may result in *instruction corruption* [13], where an instruction different from the intended one is executed. An instruction is *skipped* when the original instruction is mutated into a different one, with no additional side effects other than *not* executing the original instruction. The target will crash when the introduced mutation yields an *illegal instruction* or one that performs an *illegal memory* access. Faults affecting hardware that does not execute software, corrupt hardware that is not directly involved in software execution. This includes configuration registers, buses transferring data and logic states. Faults affecting such hardware may also have an indirect effect on software execution. For example, software execution may depend on values stored in configuration registers. This paper focuses on faults that directly affect the execution of software. We expect those faults to be the most effective for privilege escalation in the *Linux OS*.

## III. FI Setup

We created a V-FI test bench using commercially available tooling which consists of both software [14] and hardware [15]. The hardware used to provide an arbitrary voltage signal (including glitches) to the target consists of a Field-Programmable Gate Array (FPGA) and Digital-to-Analog Converter (DAC). The DAC is able to provide an arbitrary voltage signal between -4 V and 4 V with a 4 nanosecond pulse resolution. The voltage signal is used to provide power to the power domain that powers the ARM Cortex-A9 processor. This power domain powers the processor separately from the rest of the SoC.

Several modifications to the *Target* are required in order to perform V-FI. To summarize:

- **Power cut –** A power cut is created by removing a resistor bridge, in order to disconnect the original on-board power supply from the chip. This allows controlling the voltage supply to the target.
- **Trigger –** A General Purpose Input/Output (GPIO) signal is to define the *Attack Window* and can be used to time the attack. This signal, or trigger, is set before the operation under attack and unset after. It is exposed on the Printed Circuit Board (PCB) of the *Target* and can be controlled using a hardware register.
- **Reset –** A solid state relay is used to switch the external power supply whenever the *Target* enters into an unrecoverable erroneous state.
- **Capacitance –** The *Target*'s capacitance impacts the effectiveness of the injected glitch when performing V-FI. Therefore, the capacitance is kept at a minimum by removing all the bypass capacitors placed on the PCB for stability.

Several glitch parameters are configurable when performing V-FI:

- **Normal Voltage –** The voltage provided to the *Target* when no fault is injected. Not providing sufficient voltage results in a non-functional target, while an excessive voltage can potentially damage the target.
- **Glitch Voltage –** The voltage provided to the target when a fault is injected.
- **Glitch Length –** The amount of time the glitch voltage is supplied to the target instead of the normal voltage.
- **Glitch Delay –** The amount of time between the trigger is set and the glitch is injected.

Effective parameters are typically identified using a divide and conquer [12] approach where the glitch parameters are randomized.

## IV. Characterization

The *Target*'s robustness against V-FI is determined using a two-phase approach. The first phase targets code executed by the *Boot Loader* and the seconds phase targets code executed by the *Linux Kernel*. The code executed by the *Boot Loader* is easier to target as the full initialization of the *Linux OS* is not required which allows us to perform more experiments in less time. The executed test code is the same for both phases.

The *for* loop is written in C and compiled with GCC with default optimizations enabled. The test code executes in total 10000 loop iterations after which the counter value is printed on the serial interface. Therefore, the test code prints *0x2710* when no glitch is injected. Based on the counter value it is possible to determine if the expected behavior of the test code can be altered using V-FI. Other results are expected as well and classified as follows:

- **Expected –** The counter value matches the expected value and therefore the injected glitch has no impact on the test code's intended behavior.
- **Success –** The counter value is different and therefore the injected glitch alters the test code's intended behavior.
- **Crash –** The *Target* crashes. On ARM this results in a processor exception [16] after which the exception handler is executed if set up.
- **Mute –** The *Target* mutes. The injected glitch alters the test code's intended behavior but execution cannot continue and *Target* mutes.

- **Reset –** The *Target* resets. The injected glitch alters the test code's intended behavior but it results in a reset as well.

The glitch strength is often too strong for *mutes* and *resets*. The characterization of the *Boot Loader* and *Linux Kernel* is done using the parameters shown in Table I.

Table I: INITIAL GLITCH PARAMETERS

| Parameter | Value |
|---|---|
| Normal Voltage | 1.2 V |
| Glitch Voltage | Random between -4.0 V and 0.0 V |
| Glitch Delay | Target dependent |
| Glitch Length | Random between 0 ns and 1000 ns |

All parameters are identical both for the characterization of the test code executed by the *Boot Loader* and *Linux Kernel* except the *Glitch Delay*. This parameter is randomized differently depending on the characterization performed.

## A. Boot loader

The *Target*'s default *Boot Loader* (U-Boot) is modified to integrate the test code. The glitch is timed using a GPIO-based trigger as described in Section III. The GPIO control register is directly accessible by the *Boot Loader*. The attack window is roughly 15 µs. Therefore, the *Glitch Delay* is set between 5 µs and 10 µs to guarantee that the injected glitch only hits the loop.

In 17 hours we perform 27383 experiments for which the results are summarized in Table II. An experiment is classified *Expected* when the counter value matches the expected value and *Successful* when the counter value is different. All other cases are classified as *Other*.

Table II: BOOT LOADER CHARACTERIZATION

| Classification | Amount | Percentage |
|---|---|---|
| Expected | 14961 | 54.6361 % |
| Successful | 360 | 1.3147 % |
| Other | 12062 | 44.0456 % |

The plot of *Glitch Voltage* and *Glitch Length* for the first 1000 experiments is shown in Figure 1. The *Expected* experiments are plotted as $G$, the *Successful* experiments are plotted as $R$, while $Y$ is used for the experiments classified as *Other*. The plot shows a relationship between the *Glitch Voltage* and *Glitch Length*, indicated by the $R$ area between the $G$ and $Y$ areas. This means that either the *Glitch Voltage* or the *Glitch Length* can be fixed to decrease the total glitch parameter search space, significantly increasing the success rate.

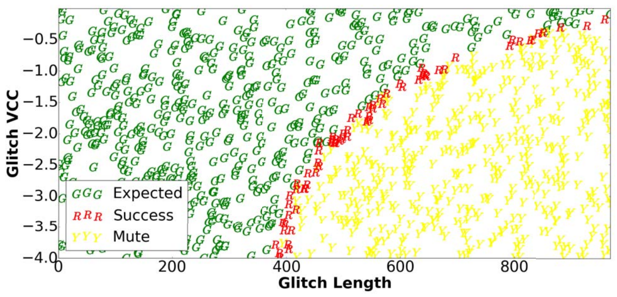

Figure 1. Boot loader characterization

## B. Linux Kernel

This characterization step is performed within the *Linux OS* which is initialized in roughly 35 seconds after power-on reset. A consistent time penalty applies each time the target resets or a reset is required, drastically increasing the time required to perform the experiments. A Linux Kernel Module (LKM) is used to integrate the test code in the *Linux OS*. This LKM implements the same C loop as used in Section IV-A. The control registers of the GPIO signal are mapped into the LKM's accessible address space in order to set the trigger up and down. The time between the trigger goes up and down is roughly 20 µs. A Linux device file is exposed to Linux user space in order to start the test code from a Linux application.

In 65 hours we perform 16428 experiments with the glitch parameters listed in Table I. The results are summarized in Table III in similar fashion as done in Section IV-A.

Table III: LINUX OS CHARACTERIZATION

| Classification | Amount | Percentage |
|---|---|---|
| Expected | 8423 | 51.2722 % |
| Successful | 781 | 4.7541 % |
| Other | 7224 | 43.9737 % |

The plot of the *Glitch Voltage* and *Glitch Length* for the first 1000 experiments is shown in Figure 2. A relationship similar to the one described in Section IV-A is present between the *Glitch Voltage* and *Glitch Length*. The $R$ area between the $G$ and $Y$ areas shows similarities as well.

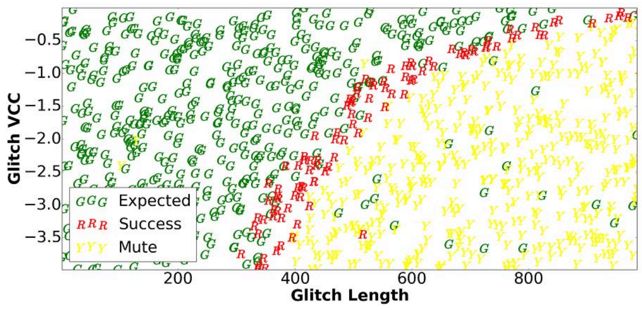

Figure 2. Linux Kernel characterization

## C. Conclusion

The two characterization step show we are able to alter the behavior of code executed by the *Boot Loader* and *Linux Kernel*. This provides enough assurance to move to the next step.

## V. Attacking Test Code

The test code targeted in Section IV-A and Section IV-B executes instructions in a loop. Therefore, the glitch timing is not so relevant as long as it hits any of the loop iterations. Most realistic attacks target a specific code construction, requiring a more accurate timing for the glitch injection. This third characterization step explores the feasibility of such condition against the *Linux OS* executed by the *Target*. The test code is implemented in a LKM that accepts three arguments from the Linux application: *int cmdid*, *int length* and *char *pointer*. The *length* argument is verified by the LKM to be smaller than *0x100*. The related code is listed in Figure 3. When the check is invalid, the LKM returns $-1$ to the Linux application. When the check is valid, the LKM returns 0 and the passed arguments to the Linux application as well.

```
if(cmd.len > 0x100) { return -1; }
```

Figure 3. Length check

The Linux application sends a valid *cmdid* and *pointer* argument to the LKM. The *length* argument is set to $0x1234$ which results in an invalid check. The expected behavior of the LKM is that $-1$ will be returned. A successful bypass of the check is identified based on the return value and the returned arguments.

In 19 hours, we perform 16315 experiments with the glitch parameters summarized in Table IV. The *Glitch Voltage* is fixed to -4V as the characterization showed a relationship between the *Glitch Voltage* and *Glitch Length*. Additionally, these values are combined so that the threshold where changes in the *Target*'s behavior are observed is barely reached. This is to minimize the *Linux OS* initialization penalty after power-on reset, achieving more successful experiments in less time.

Table IV: Initial glitch parameters

| **Parameter** | **Value** |
|---|---|
| Glitch Delay | Random between 0 µs and 8 µs |
| Glitch Length | Random between 250 ns and 300 ns |

The results are summarized in Table V. Note, a differentiation is made here between *Crash* and *Other*. The experiments classified as *Crash* are either crashes of the Linux application or *Linux Kernel*. All other experiments that affect the target's behavior negatively are classified as *Other*.

Table V: Length check results

| **Classification** | **Amount** | **Percentage** |
|---|---|---|
| Expected | 12247 | 75.0659 % |
| Successful | 5 | 0.0306 % |
| Crash | 2172 | 13.3129 % |
| Other | 1891 | 11.5906 % |

The plot of *Glitch Length*, *Glitch Delay* for the first 500 experiments is shown in Figure 4. All *Successful* results are plotted, while less results are shown for the non-successful experiments for better visualization.

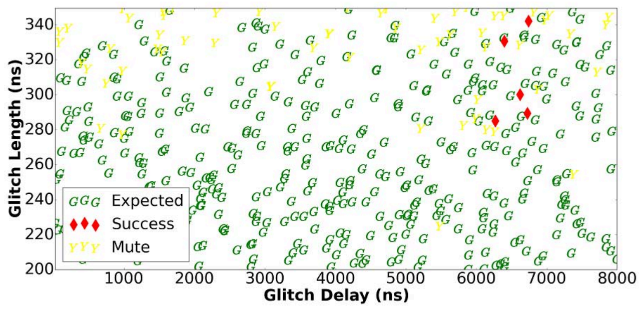

Figure 4. Bypassing length check

The plot shows that *Successful* experiments are present in a specific time window from 6.2 µs to 6.8 µs which is expected since we target specific code. The success rate within this window is 0.41%.

To conclude, this characterization step shows we are able to alter the intended behavior of a specific code construction executed by the *Linux Kernel* which provides enough assurance to move on to the next step.

## VI. Attacking Linux

This section describes three attacks where standard *Linux OS* functionality is targeted to escalate privileges using V-FI. We assume an attacker has physical access to the target and is already is able to run arbitrary Linux applications in an unprivileged context. A Linux application is used to mount each attack. Synchronization is achieved using a GPIO-based trigger for which privileged execution is required. The Linux application is started with the needed privileges, which are dropped before the glitch is injected. An attack will not have this luxury, therefore all attacks are verified to be applicable also without executing privileged code. This is accomplished by using a communication-based trigger, for which strategies are described in Section VII. The extra effort for reproducing an identified attack with such trigger is minimal.

All three attacks are performed with the *Normal Voltage* fixed to 1.2V and the *Glitch Voltage* fixed to -4.0V. The *Glitch Length* and *Glitch Delay* are randomized and different for each attack. For each attack a conservative *Glitch Voltage* and *Glitch Length* combination is used to minimize the effects of the *Linux OS* initialization time penalty.

### A. Attack 1: Open /dev/mem

The Linux system call *open* [17] is used to open a file or device. Opening the file */dev/mem* allows accessing physical addresses from Linux user space. Only Linux applications with the right *capabilities* [18] are allowed to open this specific file. After the file is opened successfully, the *mmap* system call can be used to map physical memory in the Linux application's address space. No specific capabilities are required to perform this syscall. The Linux application can then read and write physical memory, including the Linux Kernel, which allows compromising the *Linux OS* [19].

A pseudo representation of the Linux application's functionality used to mount this attack is shown in Algorithm 1. A successful attack is identified by means of the file descriptor returned by *open*, which is used for mapping physical memory with *mmap*. A read is finally performed from the returned pointer which should contain a known expected value.

**Algorithm 1** Open syscall

```
$r1 \leftarrow 2$
$r0 \leftarrow "/dev/mem"$
$r7 \leftarrow 0x5$
$swi\ 0$
if $r0 == 3$ then
    $address \leftarrow mmap(...)$
    $printf(*address)$
end if
```

In 17 hours, we perform 22118 experiments for which the results are shown in Table VI. The *Glitch Delay* is randomized between 0 µs and 30 µs. The *open* system call on */dev/mem* is expensive and therefore the time for the trigger to go up and down is relatively long.

Table VI: ATTACK 1 RESULTS

| Classification | Amount | Percentage |
|---|---|---|
| Expected | 20561 | 92.9605 % |
| Successful | 6 | 0.0271 % |
| Crash | 1547 | 6.9943 % |
| Other | 4 | 0.0181 % |

The plot of *Glitch Length* and *Glitch Delay* for the first 500 experiments, along with all the *Successful* experiments, is provided in Figure 4.

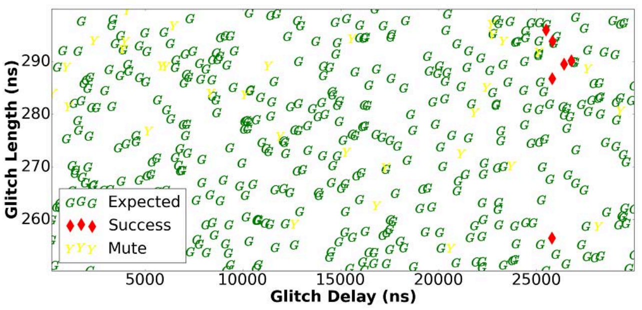

Figure 5. Attack 1 plot

The plot shows that the *Successful* experiments are present within a specific time window between 25.5 µs and 26.8 µs. This is expected as a specific code construction is targeted. The success rate is 0.53 % when the glitch parameters are tuned accordingly.

To conclude, this attack shows that we are able to bypass the restrictions enforced by the Linux *open* system call. This allows mapping physical memory from an unprivileged context which leads to a compromised *Linux OS* on a standard embedded device.

### B. Attack 2: Controlling PC

In [13] the authors show that it is possible to load controlled data into the processor's Program Counter (PC) using V-FI. This attack vector is used to take control over the target's execution during boot. In this section we determine the feasibility of this attack vector for taking control over the Linux Kernel. The Linux application executes all system calls between 0 and 255 randomly. This is done as the authors showed in [13] that the probability for loading an attacker controlled PC value is code dependent. The execution of different system calls allows exploration of different code constructions. To increase the probability, all unused processor registers are set to the value $0x41414141$ before executing the *swi* instruction. These values propagate at various locations in memory as they need to be preserved by the *Linux OS* when returning to the Linux application. A successful attack is identified when the $0x41414141$ is loaded into the PC register of the processor. This crashes the Linux Kernel. A pseudo representation of the Linux application's functionality is shown in Algorithm 2.

**Algorithm 2** Linux user space code

```
$r0 \leftarrow r1 \leftarrow r2 \leftarrow r3 \leftarrow r4 \leftarrow r5 \leftarrow 0x41414141$
$r6 \leftarrow r8 \leftarrow r9 \leftarrow r10 \leftarrow r11 \leftarrow r12 \leftarrow 0x41414141$
$r7 \leftarrow getRandom(0, 255)$
$swi\ 0$
```

The exception handler, executed after the target crashes, prints the state of the Linux Kernel at the crash moment. This is used to identify *Successful* experiments. Several lines from the exception handler printout, in case of a *Successful* experiment, are shown in Figure 6. Most importantly, the PC register is set to *0x41414141*, accounting for the crash.

```
[   23.601442] Unable to handle kernel
               paging request at
               virtual address 41414140..
[   23.683129] pc : [<41414140>]
               lr : [<c01ff158>]
[   23.858611] Code: bad PC value..
```

Figure 6. Linux Kernel exception log

In 14 hours, we perform 12705 experiments for which the results are shown in Table VII. The *Glitch Delay* is randomized between 0 µs and 3 µs. Two Linux system calls result in loading the desired value in the PC register: *sys_getgroups* and *sys_prctl*. Other system calls may be vulnerable too but this is not further explored in this paper.

Table VII: ATTACK 2 RESULTS

| Classification | Amount | Percentage |
|---|---|---|
| Expected | 11306 | 88.9886 % |
| Successful | 9 | 0.0708 % |
| Crash | 1386 | 10.9091 % |
| Other | 4 | 0.0315 % |

The system call to *sys_getgroups* is then selected for further experiments. The plot of *Glitch Length* and *Glitch Delay* for the first 500 and all *Successful* experiments is shown in Figure 4.

The plot shows that the *Successful* experiments are spread out in a larger time window than previous experiments. It is likely that multiple code constructions are vulnerable and lead to similar behavioral changes in the *Target*. Nonetheless, most *Successful* experiments are present in a window between 2.2 µs and 2.65 µs after the trigger. The success rate within this window is 0.63%.

To conclude, this attack shows that it is possible to load arbitrary values in the processor's PC register. This allows fetching instructions from an unintended address during privileged code execution. Additional effort is required to compromise the *Linux OS* on a standard embedded device.

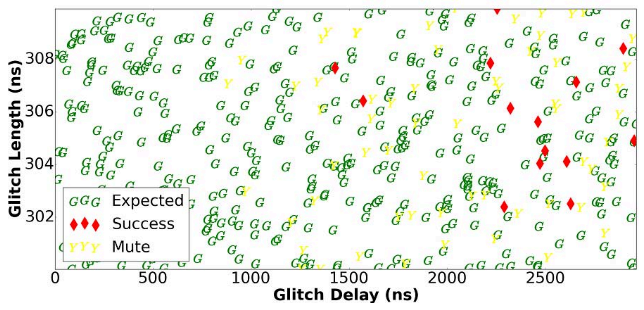

Figure 7. Attack 2 plot

### C. Attack 3: Root shell

The Linux system call *setresuid* is used to set the real, effective and saved user of a Linux application [20]. Effectively, *setresuid* allows a Linux application to continue executing as another user, including the privileged root user. This is done by setting the arguments of the *setresuid* system call to 0. The *Linux OS* prevent this practice for unprivileged processes. Such restriction is an interesting target for a FI attack. A similar Linux application to the one described in Section VI-B is used. All unused registers are set to 0 to increase the probability of a successful attack. This works as the *Linux OS* uses 0 as the return value for a function's successful execution. This is likely to increase the probability of the function failing due to our invalid input to return 0. A pseudo representation of the Linux application's functionality is shown in Algorithm 3.

**Algorithm 3** Executing a root shell

```
$r0 \leftarrow r1 \leftarrow r2 \leftarrow r3 \leftarrow r4 \leftarrow r5 \leftarrow 0$
$r6 \leftarrow r8 \leftarrow r9 \leftarrow r10 \leftarrow r11 \leftarrow r12 \leftarrow 0$
$r7 \leftarrow 0xd0$
$swi\ 0$
if $r0 == 0$ then
    $system("/bin/sh")$
end if
```

In 21 hours, we perform 18968 experiments for which the results are summarized in Table VIII. The *Glitch Delay* is randomized between 0 µs and 5 µs.

Table VIII: ATTACK 3 RESULTS

| Classification | Amount | Percentage |
|---|---|---|
| Expected | 15016 | 79.1649 % |
| Successful | 22 | 0.1160 % |
| Crash | 3919 | 20.6611 % |
| Other | 11 | 0.0580 % |

The *Successful* experiments are verified using the Linux command *whoami* which indicates the current user. The output of this command is *root* for a *Successful* experiment. The plot of *Glitch Length* and *Glitch Delay* for the first 500 and all *Successful* experiments is shown in Figure 8. The plot shows that the highest density of *Successful* experiments is happening within the time window between 3.14 µs and 3.44 µs. The success rate is 1.3 % when the *Glitch Delay* is set to this specific window.

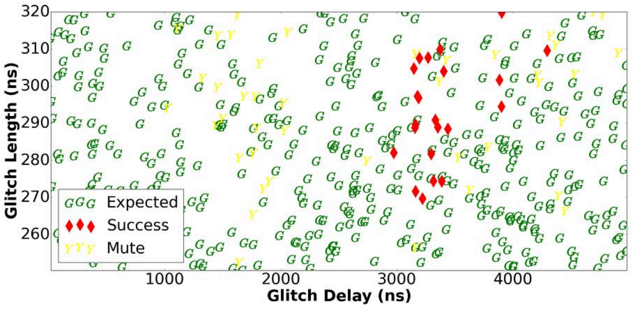

Figure 8. Attack 3 plot

To conclude, this attack shows that it is possible to bypass the restrictions implemented by the *setresuid* Linux system call. Execution of a *root shell* is then possible using the *system* function which leads to compromised *Linux OS* on a standard embedded device.

## VII. Timing

Timing is essential for FI attacks as specific code constructions are targeted. A glitch will only generate the desired fault when injected at the right moment in time. In Section IV a stable trigger is implemented using a GPIO signal by executing privileged code which is an unrealistic scenario. In this section we describe two methods to create a trigger without executing privileged code.

**Communication –** All embedded devices implement interfaces to communicate with the outside world such as *Serial* interfaces and *Ethernet* interfaces. The data transferred over these communication interfaces can be used for glitch timing. For example, it is possible to print a specific string from the Linux application just before the moment the glitch must be injected. Unfortunately, these interfaces are often buffered, therefore the communication may be observed after the desired injection time. This can be resolved by inserting a delay between the moment the communication is started and the moment the glitch should be injected. The amount of delay required can be determined using an open or compromised reference platform. This method is used for all attacks described in Section VI to verify their applicability in a realistic setting.

**Side-channels –** All embedded devices leak information through side-channels such as *Power Analysis* and *Electromagnetic Radiation* [21]. These are typically used to break cryptographic algorithms. However, these side-channels can be used to implement a trigger as well. A specific pattern of instructions can be used to generate an unique pattern in the power consumption, which can be detected using dedicated tooling. For example, the authors of [12] use a FPGA based solution for generating a trigger based on power consumption. The power consumption is tightly coupled to the software executed by the target. This results in a stable trigger where jitter is minimal.

## VIII. Mitigation

Small code bases can be effectively hardened to lower the probability of FI attacks [22], but this is challenging for larger code bases. The *FI Attack Surface* is simply too large for operating systems like the *Linux OS*. All privileged code executed by such operating system is a potential FI attack vector. Therefore, other measures must be taken to lower the probability and impact of a successful FI attack.

- Hardware countermeasures are required to deflect FI attacks on today's embedded technology. Proposals are given by the authors of [23] and [24]. These mitigations are not yet adopted by today's embedded technology, rendering FI attacks applicable to most embedded devices.
- A FI aware security design helps minimizing the impact of a successful FI attack. All assets should not be protected by a single check or feature. Checks should be implemented on multiple levels so that a *multi-glitch* is required to compromise the protected asset. To further increase the attack complexity, additional privilege levels can be introduced like provided by ARM TrustZone [25] *Trusted Execution Environment (TEE)*. A TEE should include a small *Trusted Code Base (TCB)* with a smaller FI attack surface. Additionally, the most important assets (like cryptographic keys) should be hidden from software using hardware cryptographic engines. This requires an attack to perform side-channel analysis [21] to recover the keys.
- The standard *Linux OS* can be hardened using standard features (e.g. *SELinux* [26]) and 3rd party security patches (e.g. GRSEC [27]) to increase the complexity of exploiting logical vulnerabilities. These features may have an impact on FI attacks as well.

One must realize that the mitigations may be targeted using FI as well. Therefore, they must rely on a single operation for the same reason as the code they protect as well: a single glitch is simpler to pull of than a multi glitch. Additionally, the effectiveness of the mitigations can only be determined with some assurance after actual FI testing is performed.

## IX. Conclusion

Three attacks are demonstrated where privileges are escalated in the *Linux OS*, executed on an ARM Cortex-A9 based SoC, using V-FI. An attacker with physical access to the target, and the required tooling, can take full control of the *Linux OS* without the presence of a logical vulnerability. The attacks described in this paper only applies to targets whose hardware is susceptible to FI, which holds up for most standard embedded technology of today. The success rate is sufficiently high for the attacks to be repeated within a reasonable amount of time. Decreasing the probability for a successful FI attack is challenging on complex code bases using traditional FI countermeasures. Therefore, other mitigations must be implemented to minimize the impact of such attack. The attack complexity can be increased using *Linux OS* hardening features. More importantly, the target's most important assets should not be exposed once the operating system is compromised which can be accomplished using additional privilege levels and dedicated hardware features. More concretely, a single glitch should never result in a full compromise of the target.

## REFERENCES

[1] Microsoft. Coordinated Vulnerability Disclosure. https://technet.microsoft.com/en-us/security/dn467923. [Online; accessed 2017].

[2] Hackerone. Bug Bounty Programs. https://hackerone.com/bug-bounty-programs. [Online; accessed 2017].

[3] Alessandro Barenghi, Guido M. Bertoni, Luca Breveglieri, and Gerardo Pelosi. A fault induction technique based on voltage underfeeding with application to attacks against aes and rsa. *J. Syst. Softw.*, 86(7):1864–1878, July 2013.

[4] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[5] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. *CARDIS*, 2012.

[6] Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. *FDTC*, 2014.

[7] Raghavan Kumar, Philipp Jovanovic, and Ilia Polian. Precise fault-injections using voltage and temperature manipulation for differential cryptanalysis. In *2014 IEEE 20th International On-Line Testing Symposium, IOLTS 2014, Platja d'Aro, Girona, Spain, July 7-9, 2014*, pages 43–48, 2014.

[8] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptology ePrint Archive*, 2016:810, 2016.

[9] Robert Van Spyk Rajesh Velegalati and Jasper van Woudenberg. Electro magnetic fault injection in practice. *International Cryptographic Module Conference (ICMC)*, 2013.

[10] Philippe Maurine. Techniques for em fault injection: Equipments and experimental results. *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2012.

[11] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. *CHES*, 2002.

[12] Federico Menarini Jasper G. J. van Woudenberg, Marc F. Witteman. Practical optical fault injection on secure microcontrollers. *FDTC*, 2011.

[13] N. Timmers, A. Spruyt, and M. Witteman. Controlling pc on arm using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35, Aug 2016.

[14] Riscure. Spider – Quick Start Guide (QSG 1.0). https://www.riscure.com/documents/spider-qsg-released.pdf. [Online; accessed 2017].

[15] Riscure. Inspector FI. https://www.riscure.com/security-tools/inspector-fi/. [Online; accessed 2017].

[16] ARM. About processor exceptions. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/Cihbbbcj.html. [Online; accessed 2017].

[17] Linux. Linux Programmer's Manual Open(2). http://man7.org/linux/man-pages/man2/open.2.html. [Online; accessed 2017].

[18] Linux. capabilities - overview of Linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html. [Online; accessed 2017].

[19] Anthony Lineberry. Malicious Code Injection via /dev/mem. https://phra.gs/security/malicious-code-injection-via-dev-mem.pdf. [Online; accessed 2017].

[20] Linux. Linux Programmer's Manual SETRESUID(2). http://man7.org/linux/man-pages/man2/setresuid.2.html. [Online; accessed 2017].

[21] Francois-Xavier Standaert. Introduction to Side-Channel Attacks. http://perso.uclouvain.be/fstandae/PUBLIS/42.pdf. [Online; accessed 2017].

[22] M. Witteman. Secure Application Programming in the Presence of Side Channel Atacks. https://www.riscure.com/benzine/documents/Paper_Side_Channel_Patterns.pdf. [Online; accessed 2017].

[23] David El-Baze, Jean-Baptiste Rigaud, and Philippe Maurine. An embedded digital sensor against EM and BB fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*, pages 78–86, 2016.

[24] Mario Werner, Erich Wenger, and Stefan Mangard. *Protecting the Control Flow of Embedded Processors against Fault Attacks*, pages 161–176. Springer International Publishing, Cham, 2016.

[25] ARM. ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone. [Online; accessed 2017].

[26] Michael Wikberg. Secure computing: SELinux. http://www.tml.tkk.fi/Publications/C/25/papers/Wikberg_final.pdf. [Online; accessed 2017].

[27] Brad Spengler. At ARMs Length Yet So Far Away. https://grsecurity.net/at_arms_length_h2hc_2013.pdf. [Online; accessed 2017].