QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks

Andrea Höller*, Armin Krieg, Tobias Rauter*, Johannes Iber*, and Christian Kreiner*
*Institute for Technical Informatics, Graz University of Technology
{andrea.hoeller, tobias.rauter, johannes.iber, christian.kreiner}@tugraz.at
krieg@alumni.tugraz.at

***Abstract*—Physical attacks, such as fault attacks, pose a decisive threat for the security of devices in the Internet of Things. An important class of countermeasures for fault attacks is fault tolerant software that is applicable for systems based on COTS hardware. In order to evaluate software countermeasures against fault attacks, fault injection is needed. However, established fault injection approaches require manufactured products or hardware details (e.g. netlists, RTL models), which are not available when using COTS hardware.**
In this paper, we present a QEMU-based fault injection platform that supports commercial COTS processors that are widely-used in the embedded domain. This framework allows a system-level analysis of software countermeasures by featuring the simulation of high-level hardware faults targeting, for example, memory cells, register cells, or the correct execution of instructions. The framework supports the generation of realistic fault attack scenarios. We illustrate the practicability of the approach by presenting two exemplary use cases.

I. Introduction

The trend towards the Internet of Things (IoT) offers connectivity between built-in sensors, field operation devices, and cloud systems, covering a variety of applications, including medical, home automation, energy, transportation, environmental monitoring, etc. One of the most challenging topics in such an interconnected world of miniaturized systems and sensors relate to security and privacy aspects. In contrast to traditional server farms, it is easy to gain physical access to IoT devices. This increases the vulnerability to physical attacks such as side-channel or fault attacks. While, these attacks already play an important role in the development of smart cards, there are only few studies addressing physical attacks of software-intense IoT systems. Aside from designing secure protocols and mechanisms, researchers have to work on improving security aspects of software implementations.

At the same time, embedded systems have to satisfy ever-growing demands for high computing performance and cost-efficiency. Furthermore, small start up companies are increasingly the producers of innovative IoT applications. Since their resources are limited, they typically realize systems by implementing only the software system and building the hardware platform with third-party commercial off-the-shelf (COTS) components. Unfortunately, hardware components are becoming increasingly vulnerable to faults due to the continuous shrinking of the transistor sizes [1]. Thus, a major concern of designers of dependable systems are operational faults that reduce the execution reliability. Furthermore, intentional faults caused by malicious attackers that try to put the target system into an unintended state have to be considered. Several studies showed that fault attacks could corrupt either the data loads from the memory or the assembly instructions that are executed. Thus, weaknesses in the implementation of secure systems pose a considerable vulnerability.

To identify potential weaknesses of a system regarding fault attacks, fault injection (FI) is needed. However, traditional FI techniques focus on hardware design or hardware/software codesign [2]. There are only few methods supporting software that is executed on a COTS hardware. In order to contribute towards filling this gap, we present a user-friendly framework that allows the simulation of fault attack scenarios without requiring hardware models or external hardware equipment. This framework is based on the work described in [3]. We illustrate the practicability of the approach with two use cases that exemplify how the framework can be applied to enhance the resilience of small, but security-relevant software parts.

II. Background and Related Work

A. Fault Injection Attacks

Fault attacks examine secure systems under malfunctioning. There are numerous possibilities how to deliberately introduce faults in integrated circuits as outlined in [4]. Attackers might vary operating conditions by heating or by manipulating the power supply. Also, faults can be introduced with radiation (e.g. cosmic radiation, X-ray, light or laser beams). Furthermore, attackers can inject faults precisely at targeted signals with invasive methods, like microprobing. In software, physical faults can cause three different types of errors. They can directly affect critical data values, lead to erroneous calculations, or cause control-flow errors by changing the execution sequence of a program.

B. Software Countermeasures

When using COTS hardware, embedded system designers have only limited possibilities of protecting their system with hardware-based methods. Compared to hardware countermeasures, software-based methods offer a higher level of flexibility and do not require any modification on the underlying hardware. In [4] an overview of software countermeasures is given. Common techniques directly come from software-implemented fault tolerance techniques that are typically applied to fulfill

TABLE I: Details of fault location and modes of supported faults. Adapted from [3]

Comp.	*Target*	*Fault modes*	*Implementation*
RAM	Address decoder Memory cell or R/W logic	Bit-flip, stuck-at-x, given value Bit-flip, stuck-at-x	Changes the address according to the fault mode
CPU	Instruction decoder Condition flags	Given value Bit-flip, stuck-at-x	Replaces current instruction with a given instruction Supports the following ARM flags: carry (CF), negative or less than (NF), sticky overflow (QF), overflow (VF), and zero flag (ZF)
Register	Register cell Address decoder	Bit-flip, stuck-at-x, given value Bit-flip, stuck-at-x, given value	Changes the data of the register according to the fault mode Changes the address of the register according to the fault mode

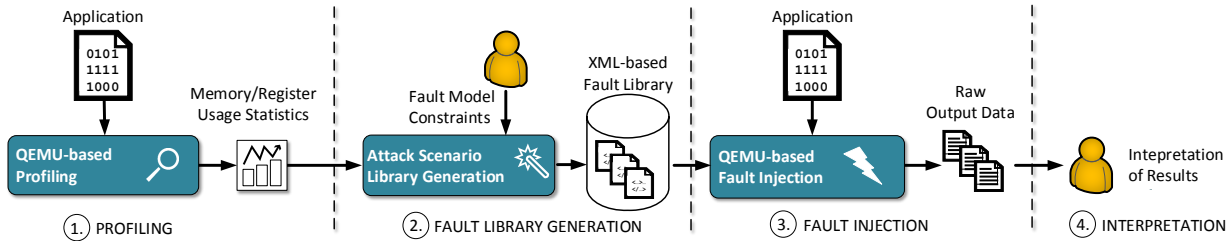

Fig. 1: Steps for simulating a fault attack scenario. First, the application is profiled to gain resource usage statistics. Then, they are used to automatically or manually create an efficient fault library representing attack scenarios. Finally, these faults are injected by the QEMU-based FI framework and the results are interpreted in order to harden the software-implementation.

high reliability requirements [5]. These principles include temporal redundancy, parity checking or checksum-based error detection. For cryptographic implementations such software-based techniques are typically applied at a function-level or algorithm level [7]. Furthermore, there are countermeasures for specific algorithms (e.g. AES [6]) and applications (e.g. Java Cards [8]). Fault attack countermeasures are defined with respect to a fault model including a theoretical set of faults an attacker could produce. FI is used to evaluate implemented fault attack countermeasures regarding the defined fault model.

C. QEMU-based Fault Injection

Many established FI approaches such as simulation-based or FPGA-based [2] techniques are not applicable for COTS processors, since they require a model of the hardware (e.g. RTL model, netlist). A promising approach supporting typical embedded COTS processor architectures are virtual FI frameworks based on emulators. A widely-used emulator for embedded systems is the Quick EMUlator (QEMU) [9]. It features the fast emulation of several CPU architectures (e.g., ARM, x86, Sparc, Alpha) on several host platforms (e.g., ARM, x86, PowerPC). QEMU is open-source and portable. Previous research has suggested methods to extend QEMU with FI capabilities to model low-level memory faults (e.g. [10], [11], [12]). The framework described in [3] not only supports memory-related faults, but also instruction execution faults. An extension of this framework targeting the reliability-assessment of applications is presented in [13]. However, as far as the authors of this paper know, all previous works on QEMU-based FI focus on fault tolerance techniques for safety purposes and do not describe the modeling of malicious faults. This paper attempts to fill this gap by presenting how to use the framework described in [3] for the evaluation of software countermeasures for fault attacks.

III. Fault Injection Procedure

We propose an evaluation of software countermeasures based on four steps as shown in Fig. 1. First, the hardware usage characteristics of the application are profiled to improve the efficiency of the FI experiments. Therefore a golden run is executed and the way to use the memory is recorded in a similar way as proposed in [14]. We implemented the collection of execution statistics before the instruction was dynamically translated by QEMU. More specifically, if the instruction performs a memory or register access, information about this access is logged. There are two statistics that are collected: the current program counter and the accessed memory/register address. Finally, there are two resulting log files: one containing statistics about the memory and one including statistics about accesses to the register.

Then, we create the fault library consisting of XML definitions based on these statistics. The FI tool allows to the injection of multiple faults at the same time. The user can create the fault library manually or use a script to generate it automatically. The number of faults that are injected in a single experiment simulating a software execution, is not limited. Thus, an XML fault definition can contain multiple faults that should appear concurrently.

The third step is to perform the FI experiments. Therefore, it is not required to adapt the source code or the binary. To analyze a software running on faulty underlying hardware with the framework, a flag is passed when executing the QEMU command providing the XML definition of the fault(s) that should be injected.

Finally, the user interprets the application outputs generated during the FI experiments. This information can support the development of efficient countermeasures.

IV. Fault Modeling Approach

Higher-level attack schemes rely on an fault model, which is an abstraction of the set of physical faults an attacker can perform [5]. Here, we present the fault model abstraction supported by the used FI framework.

A fault can be triggered by time, PC or whenever the victim component is accessed. The duration of the fault can be permanent or transient. To specify transient faults, the duration of the fault's activity can be defined. For example, this allows the attack of only certain rounds of an encryption scheme.

The framework supports the simulation of faults in the CPU and variable memory as outlined in Table I. For example, transient faults can be caused by single-event upsets and multiple-event upsets that flip one or multiple memory bits [15]. Those faults can be simulated as transient bit-flips of RAM or register cells. Furthermore, attackers could inject destructive faults, such as single-event burnout faults caused by a parasitic thyristor being formed in the MOS power transistors or single-event snap back faults triggered by a self-sustained current by the parasitic bipolar transistor in the N channel of the transistor. The fault injection framework can be used to simulate these kinds of faults by defining permanent stuck-at-0 or stuck-at-1 faults. To model memory cell faults, the framework manipulates the entire data value of a defined cell. Thus, changing a single or multiple bits of the same cell is done in the same way. The bits that should be affected can be defined with a bit mask.

A typical approach to introduce control-flow errors is to bypass instructions. For example, an access restriction mechanism based on entering a secret information (e.g., PIN, password) can be circumvented by skipping the instruction that checks the correctness of this information. Such faults can be defined by providing the address of the victim instruction and the instruction that should be executed instead. For example, this injected instruction could a jump to a branch that should be executed or it could be skipped by replacing it with a No-Operation (NOP) instruction.

V. Case Studies

A. Access Control Application and Control Flow Errors

The first use case represents a password-protected access control logic as shown in Fig. 2. For demonstration purposes, we only focus on the function `access` that implements the basic control logic. First, it calls the `check_password` function that checks whether the given password is correct. This can be done by using a secretly stored hash value of the password. If the password is correct the `check_password` function returns `1` and access to a security-critical function is granted, otherwise it shows an error message.

We used the framework to generate 23 faults representing the skipping instructions of the `access` function. Therefore several experiments were conducted, where in each experiment an instruction was replaced with a NOP. Our experiments have shown that two instructions are essential for the secure execution of the program. The first one is the instruction that jumps to the branch that is executed if the password check has failed (`beq 40001300`). The corresponding fault definition is shown in Fig. 3. If this instruction is skipped, the next

```
void access(char *given_pwd){
  bool check = check_password(given_pwd);
  if (check) do_top_secret_stuff(); //provide access
  else       printf("Wrong_Password!\n");
}
```

(a) Snipped of the C code used in the case study.

```
400012b4 <access>:
...
400012d8:   ebffffee    bl    40001298 <check_password>
400012dc:   e3500000    cmp   r0, #0
400012e0:   0a000006    beq   40001300 <access+0x4c>
400012e4:   ebffffdd    bl    40001260 <do_top_secret_stuff>
400012e8:   e594e000    ldr   lr, [r4]
...
```

(b) ARM assembler code of the case study. The highlighted instructions are vulnerable to fault attacks targeting the instruction.

Fig. 2: Use case implementing a password-protected access to a security-critical function.

```
<injection> <fault id="1">
 <comp>CPU</comp>        <target>INSTR EXECUTION</target>
 <mode>NEW VALUE</mode> <trigger>ACCESS</trigger>
 <vict_instr>0x40003b8c</vict_instr>
 <instr>0xE0D0000<!--NOP--></instr>
</fault> </injection>
```

Fig. 3: Example of a generated XML fault definition targeting the correct execution of instructions.

instruction that jumps to the function that should be protected is executed even if the password provided is incorrect. The second critical instruction evaluates the return value of the `check_password` function and sets the zero flag, if the return value was `0` indicating an incorrect password (`cmp r0, #0`). If this instruction is not executed, the zero flag is not set even if the correct password is not provided and the conditional branch to the secure function is called.

To protect the implementation regarding control flow errors, we extended the software with the jump ID countermeasure [4]. Thus, before jumping to a remote code block, the ID of the target block is assigned to a signature variable that is verified by each block. However, since the two faults that lead to a successful attack do not corrupt the flow of execution, the mechanism provided does not prevent them. Thus, our next attempt to protect the code, was to switch the branches and use the jump ID as shown in Fig. 4. This version is still vulnerable to a fault that skips the `cmp` instruction, since the zero-flag is not set and the next conditional branch (`bne 40001304`) would jump to the code that provides access to the system. However, skipping the conditional jump itself no longer leads to a security incident, since the branch that denies the access is executed thereafter. Another potential fault would be to skip the unconditional branch that is executed after the error message is prompted (`b 40012ec`). However, the jump ID is then not set correctly, since the register used to store the jump ID (`r5`) has been manipulated previously and the signature is not stored correctly.

B. Access via URLs and Memory-Related Faults

IoT services often provide a virtual infrastructure for IoT applications [16]. For example, data (e.g. Google Maps,

```
bool check = check_password(given_pwd);
if (!check)  printf("Wrong Password!\n");
else {
  jump_id=JUMPID;
  do_top_secret_stuff();
}
```

(a) C code of use case I including countermeasures.

```
400012b4 <access>:
...
400012d8:  ebffffee   bl   40001298 <check_password>
400012dc:  e3500000   cmp  r0, #0
400012e0:  1a000007   bne  40001304 <access+0x50>
400012e4:  e59f0030   ldr  r0, [pc, #48] ;40001314
...
400012f4:  e28c5001   add  r5, ip, #1
400012f8:  e5845000   str  r5, [r4]
400012fc:  eb000008   bl   40001324 <printf>
40001300:  eafffff9   b    400012ec <access+0x38>
40001304:  e3a01001   mov  r1, #1
40001308:  e5851004   str  r1, [r5, #4]
4000130c:  ebfffffd3  bl   40001260 <do_top_secret_stuff>
40001310:  eafffff5   b    400012ec <access+0x38>
...
```

(b) ARM assembler code of the enhanced implementation. The highlighted instruction in orange indicates the instruction that is still vulnerable, whereas skipping the green instruction is detected.

Fig. 4: C and assembler representation of the implemented countermeasures against instruction fault attacks.

weather information) can be obtained by using a remote API that is accessed via an URL. Such applications are often vulnerable to Bitsquatting attacks [17]. In Bitsquatting, an attacker leverages random bit-errors in memory to redirect the traffic to a domain controlled by the attacker. Therefore a domain name which has a character that differs for one-bit from the same character in the target domain is registered. Thus, there is the chance that whenever a bit-flip occurs when using the URL, the traffic is redirected to the malicious domain. Such bit-flips could be introduced deliberately with fault attacks or random bit-flips that occur occasionally are exploited. Such random bit-flips pose an increasing threat, since is expected that if there are 26 billion things connected to the internet, in 2020, there are about 665 million to 1,996 billion errors per hour across the entire IoT [18].

In this use case we evaluated an application that uses latest weather forecasts received from `weathersource.com`. We used the fault injection framework to simulate the flipping of every bit that is used to store the URL in RAM. Therefore, 128 bit-flips where simulated in four memory cells. In 42 cases, the URL resulting after the bit-flips was correct encoded and addressed (e.g., `seathersource.com`, `wdathersource.com`, `weathersnurce.com`). Fortunately, non of these URLs are registered and thus no wrong server was addressed. However, in order to prevent against potential attacks, reading and storing URLs should be protected.

VI. Conclusion

Software-based countermeasures against fault attacks offer a high flexibility and portability. To develop resilient software, potential weaknesses of the software have to first be identified with FI experiments. Based on the knowledge gained from these experiments, countermeasures can be designed.

However, many FI techniques offer only limited portability or usability, or are not applicable for COTS hardware, since they rely on detailed hardware models. Here, we presented a QEMU-based FI framework that supports COTS hardware components while providing a system-level fault model that is able to model typical common attack scenarios. Finally, we illustrated the application of the framework with a use cases showing how to evaluate small, but critical, software parts regarding the vulnerability to fault attacks.

References

[1] G. P. Saggese, N. J. Wang, T. Zbigniew, and Kalbarczyk, "An Experimental Study of Soft Errors in Microprocessors," *IEEE Micro*, 2005.

[2] A. Krieg, C. Preschern, J. Grinschgl, C. Steger, C. Kreiner, R. Weiss, H. Bock, and J. Haid, "Power And Fault Emulation for Software Verification and System Stability Testing in Safety Critical Environments," *IEEE Transactions on Industrial Informatics*, 2013.

[3] A. Höller, G. Schönfelder, N. Kajtazovic, and C. Kreiner, "FIES: A Fault Injection Framework for the Evaluation of Self-Tests for COTS-based Safety-Critical Systems," in *IEEE Microprocessor Test and Verification Workshop*, 2014.

[4] N. Theissing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive Analysis of Software Countermeasures against Fault Attacks," *Design, Automation & Test in Europe Conference*, 2013.

[5] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, 2012.

[6] F. Oboril, I. Sagar, and M. B. Tahoori, "A-SOFT-AES: Self-Adaptive Software-Implemented Fault-Tolerance for AES," in *IEEE International On-Line Testing Symposium*, 2013.

[7] M. Joye, "A Method for Preventing Skipping Attacks Marc," *IEEE CS Security and Privacy Workshops*, 2012.

[8] M. Lackner, R. Berlach, M. Hraschan, R. Weiss, and C. Steger, "A Fault Attack Emulation Environment to Evaluate Java Card Virtual-Machine Security," *Euromicro Conference on Digital System Design*, 2014.

[9] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference*, 2005.

[10] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability," *IEEE International Parallel and Distributed Processing Symposium*, 2014.

[11] F. Geissler, F. Kastensmidt, and J. Souza, "Soft Error Injection Methodology based on QEMU Software Platform," in *IEEE Latin American Test Workshop*, 2014.

[12] J. Xu and P. Xu, "The Research of Memory Fault Simulation and Fault Injection Method for BIT Software Test," *International Conference on Instrumentation, Measurement, Computer, Communication and Control*, 2012.

[13] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A Virtual Fault Injection Framework for Reliability-Aware Software Development," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2015.

[14] S. Chyek, "Collecting Program Execution Statistics with QEMU Processor Emulator," in *International Multiconference on Computer Science and Information Technology*, 2009.

[15] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, 2006.

[16] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions," *Future Generation Computer Systems*, vol. 29, no. 1, 2013.

[17] N. Nikiforakis, S. Van Acker, W. Meert, L. Desmet, F. Piessens, and W. Joosen, "Bitsquatting: Exploiting Bit-Flips for Fun, or Profit?" *International Conference on World Wide Web*, 2013.

[18] J. Schultz, "Bit Errors & the Internet of Things," 2014. [Online]. Available: http://www.darkreading.com/mobile/bit-errors-and-the-internet-of-things/d/d-id/1127914