Chaos Duck: Towards Automatic IoT Software Fault-Tolerance Analysis

(Practical Experience Report)

Igor Zavalyshyn, Thomas Given-Wilson, Axel Legay, Ramin Sadre and Etienne Rivière
ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium
{*igor.zavalyshyn,thomas.given-wilson,axel.legay,ramin.sadre,etienne.riviere*}*@uclouvain.be*

Abstract—Internet of Things (IoT) device software frequently handles sensitive data. This software has to be resistant to faults to prevent leakage and ensure data privacy and security. Source code hardening is a common way to make software fault-tolerant. However, the effectiveness and performance impact of a chosen hardening technique are not always obvious. Moreover, it becomes increasingly difficult to predict potential attack vectors and implement proper countermeasures. To assist in this task, we developed Chaos Duck, an automatic tool for IoT software fault-tolerance analysis. Chaos Duck emulates various fault types and provides statistics on their impact on software security and stability. We present a case study in which we use Chaos Duck to compare five software hardening techniques applied to the PRESENT block cipher implementation. We show that some simple hardening techniques may improve fault-tolerance, while others can instead *reduce* overall security and introduce new vulnerabilities. Our contributions are twofold: we offer a software fault-tolerance analysis tool to IoT developers seeking to make their software secure and robust, and we shed light on the efficiency of various hardening techniques.

Index Terms—fault-tolerance, fault injection, binary analysis, Internet of Things, security, encryption

I. INTRODUCTION

Millions of people worldwide use Internet of Things (IoT) devices such as smart lights, door locks and various health monitoring wearables to enhance their households and have more control over their daily lives. The nature of the data these devices operate with is often personal and sensitive. Examples of sensitive IoT data include, for instance, room/home occupancy information, door lock state updates or heart rate measurements. In order to preserve the end-user's privacy these devices usually encrypt data before transmitting it (e.g., to a local hub, a mobile phone or a cloud server). However, a single fault in the encryption logic, introduced either accidentally or intentionally by an attacker, may cause sensitive data leaks [1] as illustrated in Figure 1.

To make their devices more resistant to faults, IoT manufacturers may choose to harden the software components by adding safety logic, that aims to detect the presence of faults and minimize fault impact. Hardening can be applied to the hardware on which a given software runs [2] or to the software itself [3], [4]. While there is a variety of hardening techniques, in practice, IoT software developers rarely have a clear understanding of the real impact of a chosen hardening technique on their software's fault-tolerance and performance.
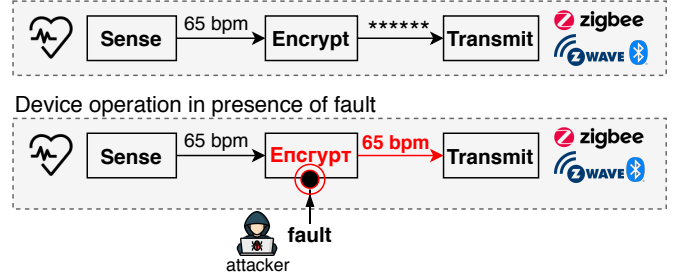

Fig. 1. Heart rate monitor's normal operation and under attack. A fault in encryption logic causes a heart rate data being transmitted in clear text.

In fact, some of the hardening techniques *may have a negative impact* and actually increase the software vulnerability [5]–[7]. Furthermore, in order to be able to implement proper countermeasures, the developers need to analyze potential attack vectors and identify sensitive code parts – a time-consuming task that requires certain expertise and resources.

To facilitate the process of IoT software hardening, we have developed and here introduce Chaos Duck – a tool for automatic software fault-tolerance analysis. Without any intervention from the developer Chaos Duck injects faults in a given software and evaluates their impact on the software's security and performance. Chaos Duck supports six different fault types ranging from instruction bit flip to branch faults, and is able to explore all the potential fault locations. Unlike other tools that require using unconventional compilers and dealing with intermediate software representations [8], [9], Chaos Duck operates on a binary level, and as a consequence requires less effort and no special expertise from the developer.

To illustrate Chaos Duck's capabilities, we perform a case study in which we compare five common software hardening techniques applied to an implementation of PRESENT – a lightweight block cipher intended to be used in IoT devices [10]. We evaluate the effectiveness of the hardening techniques on three fronts: (1) we start from evaluating their ability to prevent sensitive data leaks; then, (2) we study their general fault-tolerance and analyze the impact of each fault type; and finally, (3) we measure the impact of these hardening techniques on software performance and binary size.

The main contributions of the paper are as follows:

- We present the design and implementation of Chaos Duck tool. The results in this paper demonstrate its utility in detecting sensitive data leaks, program crashes and corruptions in control flows caused by injected faults.
- Using Chaos Duck we compare the efficacy of five common software hardening techniques and identify their strengths and weaknesses. We show that techniques exploring redundancy on a function level provide a good balance between software security and general fault-tolerance, while some of the classic hardening techniques make the hardened software even more vulnerable.

We made the source code of Chaos Duck tool as well as all of our experimental results open and publicly available [1].

The remainder of this paper is organized as follows. In Section II we explain the motivation behind this work and define a threat model. In Section III, we present the background information needed to understand the context of this work. In Section IV, we describe a high-level design of Chaos Duck and its implementation details. We follow with the description of our case study in which we perform a thorough evaluation of five hardening techniques and Chaos Duck's performance in Section V. Then, in Section VI we present the results of our case study and in Section VII we discuss the results and point out Chaos Duck's limitations and potential improvements. Finally, we highlight relevant related work in Section VIII and conclude in Section IX.

II. MOTIVATION AND THREAT MODEL

To prevent fault injection attacks manufacturers of safety- and security-critical devices often add various countermeasures into software components of these devices – a process generally known as *software hardening*. This operation assumes knowledge about the potential code vulnerabilities and attack vectors exploiting those. However, such a task is not trivial and requires high level of expertise and substantial time investments, which has a direct impact on certification of the product and its time to market. Furthermore, one harmful consequence of fault attacks is control flow disruption which may bypass some of the implemented countermeasures. Overall, it is unrealistic to expect programmers to manually investigate all possible control flow disruptions and then come up with secure and reliable countermeasures against those.

Our goal is to offer an automatic tool for IoT software fault-tolerance analysis which can emulate all the potential fault types and locations, and collect statistics on their impact on a given program's security and safety. With this tool an IoT software developer may test the efficiency of various implemented countermeasures, and depending on the results select the one that performed best.

Injecting faults via hardware is relatively difficult and requires expensive specialized hardware [11], [12]. The reproducibility and accuracy of such injections can be low depending on the fault type. We therefore concentrate on

[1] https://github.com/zavalyshyn/chaosduck

software fault injection which allows us to simulate faults at the exact locations of a program's binary. By inspecting all the potential fault locations and analyzing their impact on program behavior we can measure its tolerance to faults.

In this work we consider an attacker who does not necessarily have access to a program's source code and can only interact with instructions of a compiled binary, for instance, a device firmware blob, by having a temporary access to a given IoT device. We consider the device itself to be trusted and its software and hardware components operating normally under regular circumstances. An attacker may choose to use any of the available software [13] or hardware [12] fault injection methods available.

III. BACKGROUND

This section provides useful information for understanding the context of this work. We start from a description of fault injection attacks methods and goals, followed by an overview of different fault types.

A. Fault Injection

Securing IoT devices now is more important than ever as their numbers grow exponentially. These devices often operate with sensitive sensor data or perform a critical function and may become a primary target for attacks that put the end users' privacy, security and even safety at risk [14], [15].

Fault injection is one type of such attacks. During this attack a normal device operation mode is disrupted due to some external effect causing an unexpected behavior, i.e., *a fault*. A fault can be introduced either via hardware [16]–[18] or software [13], [19]. The impact of such a fault on the device's behavior varies considerably, ranging from no effect on its operation, to software crashes or security vulnerabilities. For instance, a simple power drop, i.e., a *glitch*, intentionally introduced by an attacker may cause data corruption or loss. Similar glitches may result in weaker data protection by disrupting the encryption logic in device firmware.

In this work we focus on faults that may introduce a security vulnerability in the device software causing the leakage of some information that was meant to remain secret. These faults are particularly dangerous when injected in software implementing secure programs such as encryption algorithms, e.g., a fault may cause the software to output a plaintext or modify a program in a way that the encryption operation is skipped completely [1].

B. Classification of Faults

We now provide an overview of the *fault models* considered in this work. We describe the logic behind each of the fault types, as well as their impact on program behavior.

A *bit flip* fault (FLP) flips a single instruction bit in a given software program. It remains the most common type of faults which may occur naturally, e.g., due to electromagnetic interference with external sources, or can be injected intentionally using specific software [20], [21] or hardware [18].

The *byte or word zeroing* faults (Z1B or Z1W, respectively) set instructions' bytes or whole word respectively to zero.

This type of fault is more likely to be related to hardware effects like an *electromagnetic pulse* (EMP), a short burst of electromagnetic energy, directed at a specific location of memory or bus, and is most effective when targeting values used in program logic. For example, the number of encryption rounds used in block ciphers. In fact, it was previously proved that skipping even a single round of encryption can have a devastating effect on security [22]–[24].

A *no operation* fault (NOP) replaces an instruction at a given address with a `nop` instruction, effectively skipping it during the execution of a program. This kind of modification may have a small effect, like skipping a variable assignment, or a large one, e.g., skipping a function call.

Finally, the goal of *conditional and unconditional branch* faults (BC and B, respectively), is to disrupt the control flow graph of a given program in a way that is beneficial for an attacker, e.g., skipping entire blocks of code such as encryption functions. This is achieved by modifying the target addresses of the branch instructions so that they point to different locations in the program's address space.

IV. CHAOS DUCK

To emulate a fault injection attack and analyze its impact on a given software binary, we developed and here introduce Chaos Duck – a tool that automatically injects faults into the binary and collects statistics on the impact of each fault type on software security and stability. To use Chaos Duck the developer only needs to provide a path to a compiled binary file and specify the architecture this binary was compiled for. The binary itself can be built with standard optimization and security hardening flags enabled, which are usually required for distribution. Chaos Duck does not require any of debug flags to be enabled since it operates on assembly code.

Chaos Duck automatically disassembles a given binary and parses its assembly code collecting information about the instructions set (e.g., address space, size and values), branch instructions' locations, types and their targets, initialization of static variables and their corresponding values, among others. This information is later used to produce *faulted* binaries – copies of the original binary each with a single injected fault.

Strategically Injecting Faults. Some faults require care in their injection, e.g., branch faults (BC or B), while others can be injected into any part of the binary, e.g., bit flip faults (FLP). Strategic fault injection requires certain knowledge of the binary from the attacker.

For branch faults (BC or B), Chaos Duck modifies the target of a branch instruction to point to a different location. For each branch instruction in the original binary multiple faulted binaries are produced each with a different target address. A new target address is picked sequentially within the program's address space as long as the destination address is within the branch range.

For NOP faults Chaos Duck checks all the possible outcomes by replacing each original instruction with a `nop` instruction. A similar approach is used for the bit flip (FLP) faults but in this case Chaos Duck performs an additional

```
$ chaosduck ./present arm -C inout.conf

Disassembling and parsing...
Number of detected branches:  37
Number of variable declarations: 25
Number of faulted binaries:  411
Running the faulted binaries...

Plaintext instead of cipher in b_at_0x105b_to_0x1628
Plaintext instead of cipher in bc_at_0x155c_to_0x1192
...
```

Listing 1. Sample ouput of Chaos Duck tool.

check *before* injecting a fault. The faulted instruction bytes with a single flipped bit are checked against an instruction format supported by a given architecture (e.g., ARM). If a flipped bit makes the instruction invalid Chaos Duck retains the original bit value and moves on to the next instruction bits flipping and checking those in turn. As a result, one original instruction results in multiple valid *'flipped'* instructions each with a single injected fault.

Next, Chaos Duck looks for variable declarations that have a numeric value of up to four bytes and 'zeros' them with Z1B or Z1W faults to produce a new faulted binary for each discovered declaration. These fault types target variables controlling the number of encryption rounds or loop counters.

Evaluating Outcomes. Depending on the injected fault the resulting binary can behave differently during its execution. Some faulted binaries may produce a result that was not expected under normal circumstances, e.g., an invalid cipher that will be impossible to decrypt. Other faulted binaries may cause a plaintext or even an encryption key to appear in the output leading to sensitive data leaks. Alternatively, faulted binaries may also terminate abruptly. Some may fail with a segmentation fault or crash with an error code, while others might get caught in an infinite loop.

Chaos Duck automatically executes all the generated faulted binaries and collects the results (stdout/stderr outputs, exit codes, and timeouts). The input arguments are provided as command line parameters or can be supplied in an external file. For software implementing an encryption algorithm Chaos Duck accepts an encryption key, plaintext and an expected cipher as input arguments. Chaos Duck then performs several checks on the binary output. First, checking whether a plaintext or an encryption key (or both) appears in any of the faulted binaries' outputs. Second, checking if the produced cipher (if any) is the expected one. In case of an invalid ciphertext, Chaos Duck records the injected fault's type and an error code (if any). Otherwise, the faulted binary is considered to have no deviations from its normal operation. In this case Chaos Duck records the execution results and marks the fault as unsuccessful.

Implementation. Chaos Duck was implemented in 2 KLoC of Python. Chaos Duck uses the Capstone[2] framework to disassemble input binaries and verify the validity of faulted instructions. Chaos Duck supports both x86 and ARM ar-

---

[2]Available at https://www.capstone-engine.org/

3

chitectures, and implements all the fault models described in Section III-B. The results of fault-tolerance analysis for a given binary are provided in a form of a CSV file with all the execution outcomes (raw form), and as a short summary with all the detected vulnerabilities, data leaks and statistics on faults success rate (report form) as shown in Listing 1.

V. CASE STUDY

To evaluate the efficacy of Chaos Duck we performed a case study in which we compared five commonly used software hardening techniques applied to an implementation of the PRESENT algorithm. We use Chaos Duck to inject faults into each hardened implementation and evaluate their impact. The goal of this study is twofold: first, we want to understand whether Chaos Duck can be a useful tool in hands of IoT developers seeking to improve the security and safety properties of their software, and, second, we are curious to know what software hardening techniques provide better fault-tolerance and are better suited for IoT scenarios.

For the purpose of this study we consider single fault injection attacks, which allows us to identify the impact of each fault type individually. Note that Chaos Duck can be configured to emulate multiple fault injections at the same time, however, this has to be done with care since it increases the number of faulted binaries and analysis time exponentially.

Next, we provide a brief overview of PRESENT block cipher used in our case study, as well as a description of five hardening techniques selected for comparison.

A. The PRESENT Block Cipher

PRESENT [10] is a block cipher that was specifically developed for low-power resource-constrained sensor devices, typical in the IoT, that due to their hardware constraints cannot use a conventional AES cipher. PRESENT is an SPN-based (substitution permutation network) block cipher with 31 rounds, a 64-bit block size and a 80- or a 128-bit key. In our case study we use a canonical size-optimized version of PRESENT implemented in C with a 80-bit key[3].

A high-level overview of PRESENT algorithm and how it can be used in a heart rate monitor is shown in Listing 2. Each of the 31 encryption rounds consists of an XOR operation to introduce a round key using S-box and permutation layers (lines 4 to 6). After that, an additional operation performs a final key XOR at line 9. A heart rate monitor runs a regular report cycle during which it obtains a new heart rate value (i.e. state), encrypts it with a generated key and transmits it to an external receiver (lines 12 to 15).

B. Hardening Techniques

To prevent potential fault injection attacks one can use various hardening techniques that aim to detect and prevent incorrect program behavior. To this end, we selected five state-of-the-art techniques commonly used to harden software implementations of cryptographic algorithms in embedded systems [25], [26]. Below we outline the key concepts of

---

[3]Availalbe at http://www.lightweightcrypto.org/implementations.php

```
1  encrypt(state,key) {
2    int round = 0;
3    while(round < 31) {
4      addRoundKey(state,key);
5      sBoxLayer(state);
6      pLayer(state);
7      round++;
8    }
9    addRoundKey(state,key);
10 }
11 reportcycle() {
12   state = sense();
13   key = generateKey();
14   encrypt(state,key);
15   transmit(state);
16 }
```

Listing 2. A pseudocode of heart rate monitor's software using PRESENT.

```
1  encrypt(state,key) {
2    int round = 0, round_dup = 0;
3    while((round < 31) &&
4    (round_dup < 31)) {
5      addRoundKey(state,key);
6      sBoxLayer(state);
7      pLayer(state);
8      round++; round_dup++;
9    }
10   if (round!=round_dup) error();
11   addRoundKey(state,key);
12 }
```

Listing 3. Classic loop hardening (CLH) technique.

these techniques applied to the PRESENT implementation as illustrated in Listing 2. For extensive discussion of each technique we refer to the cited papers.

Classic Loop Hardening (CLH). This technique has been and continues to be widely used due to its simplicity and minimal developer effort required [27]–[32]. CLH relies on duplicating the loop iteration counters and exit conditions forcing a second check at loop exit (see Listing 3). The rationale behind CLH is as follows: if an injected fault corrupts the main loop counter, the duplicated counter will still hold the correct value and will signal an error on exit condition check. We extend this technique further by once again verifying all the duplicated loop counters at the end of each code block. This is particularly important in case of block cipher implementations that often include multiple *for* or *while* loops.

Variable Duplication (VD). This technique implements redundancy at the variable level [29], [33]. Each variable is duplicated and both copies are modified in the same manner (see Listing 4), i.e., every write operation performed on the original variable is also performed on its copy. At each read operation the copies are compared for consistency: if the values do not match an error is raised. Unlike the CLH technique which concentrates on loop counters variables only, VD performs this check every time *any* variable in a given program is updated or used in conjunction with another variable.

Statement Counters (SC). This hardening technique (with minor alterations) has been previously proposed by several authors [29], [34]–[36]. SC relies on counters that are incremented and checked against the expected value after executing

```
1  encrypt(state,key) {
2    int round = 0, round_dup = 0;
3    while(round < 31) {
4      addRoundKey(state,key);
5      sBoxLayer(state);
6      pLayer(state);
7      if (round!=round_dup) error();
8      round++; round_dup++;
9    }
10   if (round!=round_dup) error();
11   addRoundKey(state,key);
12 }
```

Listing 4. Variable duplication (VD) technique.

```
1  #define DECL_INIT(cnt,x) int cnt;if((cnt=x)!=x)error();
2  #define CHECK_INC(cnt,x) cnt=(cnt==x?cnt+1:error());
3  #define RESET_CNT(cnt_while,val) (cnt_while==1||
       cnt_while==val) ? cnt_while=1 : error();
4  #define CHECK_LOOP_INC(cnt_loop,x) (cnt_loop==x) ?
       cnt_loop+=1 : error();
5  #define CHECK_LOOP_END(cnt_loop,val) if (cnt_loop!=val)
       error();
6  encrypt(state,key) {
7    DECL_INIT(enc_cnt,1);
8    CHECK_INCR(enc_cnt,1);
9    int round = 0;
10   CHECK_INC(enc_cnt,2);
11   DECL_INIT(while_cnt,1);
12   CHECK_INC(enc_cnt,3);
13   DECL_INIT(loop_cnt,0);
14   CHECK_INC(enc_cnt,4);
15   while(round < 31) {
16     RESET_CNT(while_cnt,6);
17     CHECK_LOOP_INC(loop_cnt,round);
18     CHECK_INC(while_cnt,1);
19     addRoundKey(state,key);
20     CHECK_INC(while_cnt,2);
21     sBoxLayer(state);
22     CHECK_INC(while_cnt,3);
23     pLayer(state);
24     CHECK_INC(while_cnt,4);
25     round++;
26     CHECK_INC(while_cnt,5);
27   }
28   CHECK_INC(enc_cnt,5);
29   CHECK_LOOP_END(loop_cnt,31);
30   CHECK_INC(enc_cnt,6);
31   addRoundKey(state,key);
32   CHECK_INC(enc_cnt,7);
33 }
```

Listing 5. "Statement counters (SC) technique."

```
1  reportcycle() {
2    state = sense();
3    key = {0xd3, 0xe4 ... 0xba};
4    for (int i=0; i<8; i++) {
5      copy[i] = state[i];
6    }
7
8    encrypt(state,key);
9    encrypt_dup(copy,key);
10
11   for (int i=0; i<8; i++) {
12     if (state[i]!=copy[i]) error();
13   }
14   transmit(state);
15 }
```

Listing 6. Function duplication (FD) technique.

```
1  reportcycle() {
2    state = sense();
3    key = {0xd3, 0xe4 ... 0xba};
4    for (int i=0; i<8; i++) {
5      copy[i] = state[i];
6    }
7
8    encrypt(state,key);
9    decrypt(state,key);
10
11   for (int i=0; i<8; i++) {
12     if (state[i]!=copy[i]) error();
13   }
14   transmit(state)
15 }
```

Listing 7. Decryption at place (DaP) technique.

each source code block (i.e. a function, a loop, or even a single statement). This allows the detection of attacks that disrupt the control flow of the program, e.g., by modifying the target of branch instructions, since the maliciously modified branch target would be executed in an unexpected order. We implement the variation of this technique proposed by Lalande *et al.* [35] which suggests a per-statement counter granularity for better CFG control (see Listing 5). In this case, the attack will be detected if any of the two adjacent statements in the source code are not executed in the right order. There are also additional counters for function calls, *for/while* loops or *if* blocks.

Function Duplication (FD). With this technique all sensitive program functions are duplicated and operate on the same inputs, but their outputs are stored in different variables [37] (see Listing 6). These variables are compared on function exit:

if the resulting values are different the program throws an error. The original function needs to be deterministic and have no external dependencies other than the input. FD can be further improved by changing the logic of the duplicated function so that the same fault cannot be effectively used twice.

Decryption at Place (DaP). This technique is a variation of FD and specifically targets implementations of encryption algorithms. After encrypting a given plaintext a resulting cipher is sent to a decryption function and its output is compared with the original plaintext (see Listing 7). If the encryption (or decryption) function was corrupted the resulting comparison would fail. A determined attacker would then need to corrupt the decryption function in the same way or attack the part of the program responsible for result verification.

VI. EVALUATION

To evaluate hardening effectiveness, we compare five implementations of PRESENT hardened with techniques described in Section V-B with a non-hardened (i.e. *baseline*) canonical C implementation. All implementations are compiled for the ARM architecture using the *arm-linux-gnueabi-gcc* compiler with no optimization (-O0 flag). For this set of experiments, compile-time optimization has been intentionally disabled to be able to clearly differentiate between the results of source code hardening and compiler actions. However, Chaos Duck itself makes no assumptions about the optimization level used by the programmer and can work with binaries compiled with optimizations up to level 3. We specifically target the ARM architecture since many IoT devices available on the

5

market are ARM-based. The resulting binaries accept a 64-bit plaintext and a 80-bit key as input and output a 64-bit cipher.

With Chaos Duck we apply the fault models described in Section III-B to the baseline and hardened binaries and generate *faulted* binaries, i.e., copies of the original binary with a single injected fault. For each fault model, every possible fault location is considered. We use a set of three encryption keys and three plaintexts resulting in nine executions per binary with a 3 second timeout for each. We then measure the total number of faulted binaries for baseline and hardened versions and collect statistics on fault types and their success rate. The latter serves as an indicator of hardening technique efficiency against fault injection attacks.

We also analyze the ability of the non-hardened and hardened PRESENT implementations to withstand a key recovery attack when combined with a *cryptanalytical attack* (CA). During this attack the last 31st round of encryption is skipped making it easier to extract the encryption key as part of a *differential fault analysis* (DFA) [22], [23]. We simulate such an attack by manually setting the rounds number to 30 and recording the resulting ciphers for all *key & plaintext* pairs. We then check whether any of these ciphers appear consistently in the outputs of faulted binaries for hardened and non-hardened implementations.

Finally, we measure the average execution time for baseline and hardened binaries across 10,000 executions with randomly generated *key & plaintext* pairs and with the first 200 execution results skipped to avoid caching concerns. Additionally, we record the size in bytes for all binaries.

We evaluate hardening techniques on three fronts. First, we analyze their overall effectiveness in preventing sensitive data leakage, e.g., a plaintext or an encryption key, in presence of faults. Second, we study their general fault tolerance against different types of faults, and analyze the impact of each fault type. Finally, we analyze the performance impact of each of the hardening techniques on program runtime as compared with the non-hardened version.

A. Sensitive data leakage

For each hardening technique we count the number of faulted binaries that leaked a plaintext or an encryption key in their outputs (stdout or stderr) consistently across all 9 executions. Each leak type is presented in two categories: *normal* faulted binary execution and an *interrupted* faulted binary execution based on a 3 sec timeout (marked with 'timeout'). In case of the latter, the output of the faulted binary is a (potentially) non-terminating stream of bytes which may include a plaintext or encryption key.

Table I features the results of our analysis. We observe that with hardened and non-hardened faulted binaries it was only possible to leak a plaintext, and never a key, indicating that leaking a key is a difficult task (at least when a single fault injection is considered). As for the plaintext leakage, one can see that the hardening techniques exploring SC, FD and DaP were less prone to such leakage producing less than 0.01% of the faulted binaries that leak a plaintext as compared to the

TABLE I
SENSITIVE DATA LEAKAGE IN HARDENED AND NON-HARDENED BINARIES (NORMALLY TERMINATED VS. TERMINATED BY TIMEOUT).

	Baseline	CLH	VD	SC	FD	DaP
Binaries	**51,785**	**159,253**	**296,339**	**1,508,330**	**125,666**	**135,775**
Key leak	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Plaintext leak	48 / 14	50 / 0	148 / 0	2 / 0	0 / 2	6 / 2
DFA vulnerable	2 / 0	9 / 0	2 / 0	0 / 0	0 / 0	0 / 0

TABLE II
FAULT TYPES STATISTICS FOR FAULTED BINARIES LEAKING SENSITIVE DATA (NORMALLY TERMINATED VS. TERMINATED BY TIMEOUT).

	Baseline	CLH	VD	SC	FD	DaP
Binaries	**48 / 14**	**50 / 0**	**148 / 0**	**2 / 0**	**0 / 2**	**6 / 2**
FLP	5 / 12	1 / 0	2 / 0	2 / 0	0 / 0	4 / 0
Z1B/Z1W	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
NOP	0 / 0	0 / 0	0 / 0	9 / 0	0 / 0	0 / 0
B	22 / 0	12 / 0	20 / 0	0 / 0	0 / 0	0 / 0
BC	21 / 2	37 / 0	126 / 0	0 / 0	0 / 2	2 / 2

baseline. The CLH and VD techniques were, however, less efficient. In both cases the number of faulted binaries leaking plaintext increased, sometimes significantly. For instance, for the VD technique the number of such faulted binaries tripled. This technique proved to be largely ineffective at protecting against data leaks, and instead introduced new leak opportunities. This is mainly due to the fact that along with any intermediate variables used by the encryption function, a variable containing a plaintext data was also duplicated which doubled its chances of being leaked.

Two faulted binaries of a non-hardened (baseline) PRESENT implementation were vulnerable to the DFA attack. Their 31st round of encryption was skipped consistently for any *key & plaintext* pair. A bit flip (FLP) fault was the cause in both cases: by flipping a single bit in a branch instruction regulating the encryption loop the type of branch was changed from bls (branch if less or equal to 30) to blt (branch if less than 30). Similarly, only two hardened versions of PRESENT were vulnerable, namely CLH and VD with 9 and 2 binaries respectively. The leaks were caused by the same FLP faults as in a baseline version.

Next, we analyze the type of faults causing faulted binaries to leak sensitive data (plaintext) for a baseline and five hardened versions. As seen in Table II, the vast majority of leaks were caused by branch instructions faults. This was expected as these faults often aim to cause the program to skip the execution of a sensitive function, e.g., encryption, causing a plaintext being output as is. Bit flip faults (FLP) were the second most common cause of data leaks, followed by NOP faults with just a few binaries leaking sensitive data. Z1B/Z1W faults, however, failed to cause a leak in all cases.
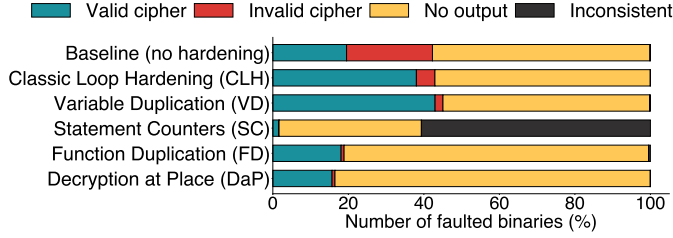
Fig. 2. Statistics on execution results in presence of faults.

Fig. 3. Statistics on failed executions.

Overall, the FD and SC techniques proved to be the most effective in protecting against sensitive data leaks. The FD technique explores redundancy of sensitive computations while SC performs a fine-grained statement-level control flow control. In contrast to other hardening techniques, FD operates with the final encryption results instead of intermediate ones, and the SC technique monitors the execution of a whole program, instead of monitoring only a single code portion.

B. Fault Tolerance

To analyze the general fault tolerance of the baseline and hardened versions we measure the percentage of faulted binaries that were unaffected by any of the faults and produced a valid cipher, then those that produced an invalid cipher, and, finally, those that crashed and produced no output. We also measure the percentage of binaries producing inconsistent results across all 9 executions with different key/plaintext pairs. Note that the binaries leaking sensitive data in their output were not considered in this experiment. The results of the analysis are presented in Figure 2.

The percentage of faulted binaries producing a valid cipher is higher, sometimes significantly, than the baseline for only two of the hardening techniques, namely CLH with 38.02 % and VD with 42.97 %. These techniques explore redundancy on a variable level and proved to be more resistant to faults as compared to other techniques, while techniques exploring redundancy on a function level (i.e. FD and DaP) were less efficient. The lowest percentage was achieved with the SC technique (only 1.64 %). This particular technique also resulted in the highest percentage of inconsistent results (60.66 %). Taking a closer look at these inconsistent results we found out that in most of the cases the binaries produced a valid cipher (71 %) but failed to do that consistently for all 9 key/plaintext pairs. At the same time, the vast majority of faulted binaries across all five hardening techniques crashed during the execution and provided no output. This was expected since in most of the cases the injected faults corrupt the program logic and raise exception errors. To have a better understanding of the true causes of these crashes we collect statistics on the error codes returned by the crashed binaries. The results are presented in Figure 3.

In case of a baseline non-hardened PRESENT version, the majority of faulted binaries crashed due to a segmentation fault, while the others were interrupted by timeout or crashed
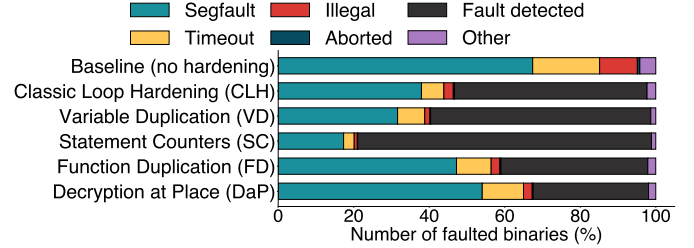
while trying to execute an illegal instruction. For the hardened binaries the situation was slightly different. While segmentation faults and timeout errors still constitute the major cause of failure, particularly for DaP and FD techniques, a significant portion of faulted binaries detected a presence of faults in their execution logic and terminated by throwing a corresponding error. The fault detection rate varies across all five hardening techniques ranging from 30% (the lowest) to 78% (the highest) for the DaP and SC techniques respectively.

The results show that depending on the requirements different hardening techniques should be used. For instance, if the program needs to be robust and output a valid cipher in majority of cases and fail otherwise, then the VD technique should be preferred. If security is more important than performance, the SC technique is a better option since it ensures the highest error detection rate.

C. Performance Analysis

All five hardening techniques have little impact on the binary size adding on average 1 KByte to the original size (16.4 KBytes), the only exception being the technique implementing statement counters (SC) that nearly doubles the size of the original binary. This technique adds two additional lines of code for each line in the original non-hardened code. In terms of runtime performance, we see no significant difference in execution times (52.4 ± 2 ms).

VII. DISCUSSION

Considering the leakage of sensitive information, we observe that none of the hardening techniques were able to prevent data leakage. In all cases Chaos Duck was able to produce at least one faulted binary leaking a plaintext. This matches with recent results on the impossibility of effective countermeasures to faults [38], but is also concerning since this kind of leakage is a dangerous vulnerability. On the other hand, recent research shows that effective hardening against specific kinds of faults is possible [38]. More broadly we observe that some of the hardening techniques were less affected by the presence of faults, namely FD, SC, and DaP. Their numbers of faulted binaries leaking plaintext was significantly lower as compared to other techniques. Both the FD and DaP techniques explore redundancy on a function level and can detect faults in the final output as opposed to techniques

that operate on the intermediate values, e.g., loop counters. However, such an approach may still be vulnerable to multiple fault injections that other hardening techniques would detect locally. The SC technique is unique in this regard since it verifies both final and intermediate computation results.

The number of faulted binaries still producing a valid cipher in presence of faults is another important parameter for analysis. Our experiments showed that the majority of faulted binaries simply failed to execute correctly, and either crashed throwing a segmentation fault or got caught in an infinite loop. In case of the VD technique, however, the percentage of faulted binaries producing a valid cipher was the highest across all other hardening techniques. The hardened code left fewer opportunities for faults to corrupt the algorithm logic. We can conclude that VD is an effective way to ensure normal operation of the binary in presence of faults.

When it comes to types of faults that contributed to sensitive data leaks there is an absolute leader – branch instruction faults (BC and B). The developers must pay attention to the way the chosen hardening technique affects the control flow graph of the program. The hardening techniques that add more branching behavior to the binary should therefore be avoided since they create more locations that can be exploited by an attacker to leak sensitive information. This was confirmed by the test results in which both VD and CLH techniques showed the highest success rate for branch instruction faults. New branches introduced by duplicating and checking global or local variables' values inadvertently increased the attack surface. On the contrary, alternative techniques, e.g., SC, FD and DaP, proved to be less affected by this type of fault.

Next, we briefly discuss the developer effort needed to implement the hardening techniques described in this paper. Some of these techniques, namely CLH, SC and VD, are prone to mistakes when implemented manually. They require specialized tools that annotate the source code automatically. In this case the developers remain oblivious to the nature of the hardening modifications and their impact on program security. Other techniques, like FD and DaP, are easier to implement and reason about.

Overall, there is a great potential in hardening techniques exploring redundancy on a function level. This granularity is in a sweet spot between the required developer effort and a desired program security when a single fault injection is considered.

In terms of performance impact of the hardening techniques, in all cases we see no significant impact on runtime performance, nor on binary size.

Finally, we discuss Chaos Duck's performance and ability to simulate various fault models. Chaos Duck proved to be a useful tool in hands of a developer seeking to improve the security properties of a software she develops. In our case study we consider an implementation of the PRESENT encryption algorithm, however any type of software that needs to meet certain security or privacy requirements (e.g., PCI SSC[4]

or ISA/IEC 62443[5] standards) can be effectively analyzed with Chaos Duck. Chaos Duck can systematically perform a fault-tolerance analysis as part of a threat modeling and risk analysis cycle (potentially as part of a standard continuous integration and development – CI/CD – process). We note, however, that the Chaos Duck prototype could be further optimized in order to reduce the time needed to explore all the potential fault space. This can be achieved, for instance, by further parallelizing its execution process on multicore CPUs or applying a worker-leader scheme in a distributed setting.

VIII. RELATED WORK

There is a significant body of work in the area of fault injection and software hardening. We will now provide a brief overview of the most relevant content describing the differences and similarities with the approach proposed here.

Fault Injection: There are many recent works that have explored fault injection, in particular on a software level [1], [8], [9], [35], [39]–[44]. Le *et al.* introduce a symbolic LLVM-based Software-implemented Fault Injection (SWiFI) evaluation framework [8]. The focus of their work is to test robustness of systems by emulating fault injections and detecting vulnerabilities. However, the proposed system works with an intermediate LLVM-IR representation of a program which limits the ability to operate on a compiled binary directly.

An alternative Symbolic Program Level Fault Injection and Error Detection Framework (SymPLFIED) is a program-level framework designed to identify potential software vulnerabilities [39]. It uses symbolic execution and model checking techniques on low level program representations, but is only able to support the MIPS architecture.

The Lazart tool can emulate a variety of fault injection attacks and allows to detect security vulnerabilities using formal methods [9]. Similarly to SWiFI framework the Lazart tool emulates control flow modifications on the LLVM-IR representation of a program which limits its application. In contrast, Chaos Duck requires no modifications of the original binary and can work with the assembly code directly.

Rivière *et al.* propose combining the Lazart tool with the Embedded Fault Simulator (EFS) [41]. They extend Lazart by adding lower level fault injection analysis that is also embedded in the chip with the program [40]. The simulation of the fault is performed on the hardware level, so the semantics of the executed program correspond to the real execution of the program. However, EFS is limited to instruction skip faults only, i.e., `nop` faults, and does not consider faults disrupting program's control flow.

A low-level approach is taken by Moro *et al.* which relies on model checking to formally prove the correctness of the proposed software countermeasures schemes against fault injection attacks [42]. Again, the focus in this case is on a very specific and limited model of fault injection that causes instruction skips and ignores other kinds of attacks.

Furthermore, the model checking is performed over the limited fragments of an assembly code.

A fault model inference focused approach is taken by Dureuil *et al.* [43]. The authors select a hardware model and then test various fault injection attacks upon it. Fault injection is limited to EEPROM faults on the ARMv7-M architecture, and the fault model is inferred from the parameters of the attack and the embedded program. The faults are emulated upon the assembly code and the results are checked with predefined oracles on the embedded program.

Given-Wilson *et al.* perform a systematic analysis of several approaches to software-based fault injection emulation and compare those with the results of hardware-based experiments [44]. In particular, they show that software based emulation approaches are able to identify locations of likely vulnerability which can then be verified by hardware experiments. Furthermore, the authors highlight the need for automating the process of fault injection vulnerability detection and integrating it into a software development cycle, which is exactly what Chaos Duck allows to do.

Overall, all available tools for fault-injection emulation and vulnerability detection are still far from being commonly used. They assume the developer to have certain expertise in statistical model checking and experience in working with low-level program representations, which is not always a case. In contrast to these tools, Chaos Duck is fully automatic, it does not require any special skills from the developer, works on a compile binary level and can be easily integrated in a standard CI/CD cycle.

Hardening Techniques: Various hardware and software hardening techniques have been proposed since the early 1970's [25], [26], [45]. The most well-known technique is the one implementing the N-Version approach [46]. In it, multiple implementations (versions) of the same algorithm are executed in parallel and their results are compared for consistency before proceeding. The computation redundancy ensures the fault-tolerance, since a fault in one version will cause an inconsistency with results from other versions. Inspired by this approach many hardening techniques implement the same approach but on the variable [29], [30], [32], [33], statement [27]–[29], function [37], or even instruction level [25], [42], [47]–[49].

Another classic countermeasure against software faults relies on using 'canary' words strategically placed in the program's memory stack by a compiler to prevent buffer overflow attacks [50]. Other techniques suggest encrypting the pointer addresses instead [51] or randomizing the address space [52], [53]. Alternative techniques propose countermeasures based on hardware and software checksums, ratification counters and baits or randomization of execution order [37].

IX. Conclusions

Faults can have a devastating effect on IoT software security, especially those that target components implementing encryption algorithms. This work presents the Chaos Duck tool for automated fault tolerance analysis and describes

our experience in using it to evaluate the efficiency of five software hardening techniques applied to an implementation of PRESENT cipher. The results show that techniques exploring redundancy on a function level such as FD and DaP strike a good balance between software safety and performance properties. At the same time, some of the classic techniques tend to make software *more vulnerable* to faults resulting in even bigger exposure or data leaks. Chaos Duck was able to efficiently inject six different fault types ranging from bit flips to branch faults, and then provide a report on detected security vulnerabilities, including sensitive data leaks, corrupted outputs, and inputs leading to error or failure states.

Overall, an automated fault tolerance analysis tool such as Chaos Duck can be a useful tool in hands of an IoT developer seeking to improve the security properties of the software she develops. We envision Chaos Duck to be used to systematically perform a fault-tolerance analysis as part of a continuous integration and development (CI/CD) cycle.

References

[1] T. Given-Wilson, A. Heuser, N. Jafri, and A. Legay, "An automated and scalable formal process for detecting fault injection vulnerabilities in binaries," *Concurr. Comput. Pract. Exp.*, vol. 31, no. 23, 2019. [Online]. Available: https://doi.org/10.1002/cpe.4794

[2] B. Sunar, G. Gaubatz, and E. Savas, "Sequential circuit design for embedded cryptographic applications resilient to adversarial faults," *IEEE Transactions on Computers*, vol. 57, no. 1, pp. 126–138, 2007.

[3] B. Gierlichs, J.-M. Schmidt, and M. Tunstall, "Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output," in *International conference on cryptology and information security in Latin America*. Springer, 2012, pp. 305–321.

[4] A. Poschmann, A. Moradi, K. Khoo, C.-W. Lim, H. Wang, and S. Ling, "Side-channel resistant crypto for less than 2,300 ge," *Journal of Cryptology*, vol. 24, no. 2, pp. 322–345, 2011.

[5] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 29–40.

[6] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 319–330.

[7] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[8] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Resilience evaluation via symbolic fault injection on intermediate code," in *Proc. of DATE*. IEEE, 2018, pp. 845–850.

[9] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *Proc. of ICST*. IEEE, 2014, pp. 213–222.

[10] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 450–466.

[11] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proc. of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[12] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2013, pp. 77–88.

[13] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: the mefisto tool," *Predictably Dependable Computing Systems*, pp. 329–346, 1995.

[14] A. Hern, "Hacking risk leads to recall of 500,000 pacemakers due to patient death fears," https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update.

[15] N. Garun, "Almost half a million pacemakers need a firmware update to avoid getting hacked," https://www.theverge.com/2017/8/30/16230048/fda-abbott-pacemakers-firmware-update-cybersecurity-hack.

[16] Y.-i. Hayashi, N. Homma, T. Sugawara, T. Mizuki, T. Aoki, and H. Sone, "Non-invasive emi-based fault injection attack against cryptographic modules," in *2011 IEEE International Symposium on Electromagnetic Compatibility*. IEEE, 2011, pp. 763–767.

[17] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of three physical fault injection techniques to the experimental assessment of the mars architecture," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 267–288, 1998.

[18] D. F. Kune, J. Backes, S. S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu, "Ghost talk: Mitigating emi signal injection attacks against analog sensors," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 145–159.

[19] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *2003 Symposium on Security and Privacy, 2003.* IEEE, 2003, pp. 154–165.

[20] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[21] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1–18.

[22] G. Wang and S. Wang, "Differential fault analysis on present key schedule," in *2010 International Conference on Computational Intelligence and Security*. IEEE, 2010, pp. 362–366.

[23] N. F. Ghalaty, B. Yuce, and P. Schaumont, "Differential fault intensity analysis on present and led block ciphers," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015, pp. 174–188.

[24] T. Given-Wilson, N. Jafri, and A. Legay, "Combined software and hardware fault injection vulnerability detection," *Innovations in Systems and Software Engineering*, 2020.

[25] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented aes: effectiveness and cost," in *Proceedings of the 5th Workshop on Embedded Systems Security*, 2010, pp. 1–10.

[26] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 404–409.

[27] G. Barbu, P. Andouard, and C. Giraud, "Dynamic fault injection countermeasure," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2012, pp. 16–30.

[28] G. Bouffard, B. N. Thampi, and J.-L. Lanet, "Detecting laser fault injection for smart cards using security automata," in *Proc. of SSCC*. Springer, 2013, pp. 18–29.

[29] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2231–2236, 2000.

[30] R. De Keulenaer, J. Maebe, K. De Bosschere, and B. De Sutter, "Link-time smart card code hardening," *International Journal of Information Security*, vol. 15, no. 2, pp. 111–130, 2016.

[31] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "Fissc: A fault injection and simulation secure collection," in *Proc. of SAFECOMP*. Springer, 2016, pp. 3–11.

[32] J. Proy, K. Heydemann, A. Berzati, and A. Cohen, "Compiler-assisted loop hardening against fault attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 1–25, 2017.

[33] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new software-based technique for low-cost fault-tolerant application," in *Annual Reliability and Maintainability Symposium, 2003*. IEEE, 2003, pp. 25–28.

[34] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, 1999.

[35] J.-F. Lalande, K. Heydemann, and P. Berthomé, "Software countermeasures for control flow integrity of smart card c codes," in *Proc. of Esorics*. Springer, 2014, pp. 200–218.

[36] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[37] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proc. of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[38] T. Given-Wilson and A. Legay, "Formalising fault injection and countermeasures," in *Proc. of ARES*, 2020.

[39] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *Proc. of DSN*. IEEE, 2008, pp. 472–481.

[40] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys, "Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks," in *Proc. of FPS*. Springer, 2014, pp. 92–111.

[41] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant, "Idea: embedded fault injection simulator on smartcard," in *Proc. of ESSoS*. Springer, 2014, pp. 222–229.

[42] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[43] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *Proc. of CARDIS*. Springer, 2015, pp. 107–124.

[44] T. Given-Wilson, N. Jafri, and A. Legay, "The state of fault injection vulnerability detection," in *Proc. of VECoS*, vol. 11181. Springer, 2018, pp. 3–21.

[45] G. C. Gilley, L. Bearnson, C. Carroll, W. Bouricius, E. Hsieh, G. Putzolu, J. Roth, P. Schneider, C. Tan, M. Hsiao *et al.*, "International symposium on fault-tolerant computing, digest of papers." Pasadena, California, March 1-3 1971.

[46] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[47] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 114–130.

[48] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a countermeasure against instruction-skip fault attacks," in *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, 2016, pp. 1–6.

[49] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 243–254.

[50] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[51] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*. USENIX Association, 2003, pp. 7–7.

[52] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *USENIX Security Symposium*, vol. 12, no. 2, 2003, pp. 291–301.

[53] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 1997, pp. 67–72.