# Brief Announcement: Effectiveness of Code Hardening for Fault-Tolerant IoT Software

Igor Zavalyshyn, Thomas Given-Wilson, Axel Legay, and Ramin Sadre

UCLouvain, Belgium

**Abstract.** Internet of Things (IoT) device software has to be resistant to faults to ensure data privacy and security. In this work, we examine five common software hardening techniques and study their impact on software fault-tolerance and security. We experimentally show that some of these techniques may improve fault-tolerance, while the others can *reduce* overall security. We offer a guideline for IoT developers seeking to make their software robust, and propose a tool for automatic software fault-tolerance evaluation.

## 1 Introduction

In order to preserve the end-users privacy IoT devices usually encrypt the sensor data before transmitting it (e.g. to the local hub, mobile phone or cloud server). However, a single fault in the encryption logic, introduced either accidentally (e.g. electromagnetic interference) or intentionally by a malicious attacker, may cause sensitive data leaks.

To make the devices more resistant to faults, IoT manufacturers often harden the software components by adding a safety logic that aims to detect the presence of faults and minimize their impact. While there is a variety of hardening techniques [6], 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.

In this paper we present a thorough IoT software hardening analysis. We select five popular and (potentially) automated hardening techniques applied to implementation of PRESENT – a widely understood IoT focused encryption algorithm. We then expose the hardened versions of PRESENT to six kinds of faults, and evaluate the effectiveness of the hardening techniques on three fronts: (1) their ability to prevent sensitive data leaks; then, (2) their general fault-tolerance and the impact of each fault type; and, finally, (3) the impact of hardening techniques on software performance and binary size.

We show experimentally that techniques exploring redundancy on a function level provide a good balance between software security and general fault-tolerance, while classic loop hardening techniques in some cases make the hardened software less fault-tolerant. To facilitate the analysis we have developed Chaos Duck – a tool for automatic software fault-tolerance evaluation. The results in this paper exploit Chaos Duck and demonstrate its utility in detecting sensitive data leaks, program crashes and corruptions in data and control flows caused by injected faults.

```
encrypt(state,key) {
  int round = 0;
  while(round < 31) {
    addRoundKey(state,key);
    sBoxLayer(state);
    pLayer(state);
    round++;
  }
  addRoundKey(state,key);
}
reportcycle() {
  state = sense();
  key = {0xd3, 0xe4 ... 0xba};
  encrypt(state,key);
  transmit(state);
}
```

**Listing 1.1.** A pseudocode of heart rate monitor's software using PRESENT.

```
encrypt(state,key) {
  int round = 0, round_dup = 0;
  while((round < 31) &&
  (round_dup < 31)) {
    addRoundKey(state,key);
    sBoxLayer(state);
    pLayer(state);
    round++; round_dup++;
  }
  if (round!=round_dup) error();
  addRoundKey(state,key);
}
```

**Listing 1.2.** Classic loop hard. (CLH).

```
encrypt(state,key) {
  int round = 0, round_dup = 0;
  while(round < 31) {
    addRoundKey(state,key);
    sBoxLayer(state);
    pLayer(state);
    if (round!=round_dup) error();
    round++; round_dup++;
  }
  if (round!=round_dup) error();
  addRoundKey(state,key);
}
```

**Listing 1.3.** Variable duplication (VD).

```
reportcycle() {
  state = sense();
  key = {0xd3, 0xe4 ... 0xba};
  for (int i=0; i<8; i++) {
    copy[i] = state[i];
  }

  encrypt(state,key);
  encrypt_dup(copy,key);

  for (int i=0; i<8; i++) {
    if (state[i]!=copy[i]) error();
  }
  transmit(state);
}
```

**Listing 1.4.** Function duplication (FD).

```
reportcycle() {
  state = sense();
  key = {0xd3, 0xe4 ... 0xba};
  for (int i=0; i<8; i++) {
    copy[i] = state[i];
  }

  encrypt(state,key);
  decrypt(state,key);

  for (int i=0; i<8; i++) {
    if (state[i]!=copy[i]) error();
  }
  transmit(state)
}
```

**Listing 1.5.** Decryption at place (DaP).

```
#define DECL_INIT(cnt,x) int cnt;
    if((cnt=x)!=x) error();
#define CHECK_INC(cnt,x) cnt=(cnt==
    x ? cnt+1 : error());
#define RESET_CNT(cnt_while,val) (
    cnt_while==1||cnt_while==val)
    ? cnt_while=1 : error();
#define CHECK_LOOP_INC(cnt_loop,x)
    (cnt_loop==x) ? cnt_loop+=1 :
    error();
#define CHECK_LOOP_END(cnt_loop,val
    ) if (cnt_loop!=val) error();
encrypt(state,key) {
  DECL_INIT(enc_cnt,1);
  CHECK_INCR(enc_cnt,1);
  int round = 0;
  CHECK_INC(enc_cnt,2);
  DECL_INIT(while_cnt,1);
  CHECK_INC(enc_cnt,3);
  DECL_INIT(loop_cnt,0);
  CHECK_INC(enc_cnt,4);
  while(round < 31) {
   RESET_CNT(while_cnt,6);
   CHECK_LOOP_INC(loop_cnt,round);
   CHECK_INC(while_cnt,1);
   addRoundKey(state,key);
   CHECK_INC(while_cnt,2);
   sBoxLayer(state);
   CHECK_INC(while_cnt,3);
   pLayer(state);
   CHECK_INC(while_cnt,4);
   round++;
   CHECK_INC(while_cnt,5);
  }
  CHECK_INC(enc_cnt,5);
  CHECK_LOOP_END(loop_cnt,31);
  CHECK_INC(enc_cnt,6);
  addRoundKey(state,key);
  CHECK_INC(enc_cnt,7);
}
```

**Listing 1.6.** Statement counters (SC) hardening.

## 2 Case Study and Methodology

In this work we evaluate the efficiency of five common software hardening techniques applied to the implementation of PRESENT [3] – a block cipher specifically designed for low-power resource-constrained IoT devices. Listing 1.1 illustrates an implementation of PRESENT used in a heart rate monitor software.

We consider five popular hardening techniques in this case study: *classic loop hardening* (CLH) [2] (Listing 1.2); *variable duplication* (VD) [4] (Listing 1.3); *function duplication* (FD) [1] (Listing 1.4); *decryption at place* (DaP) [1] (Listing 1.5); and *statement-based counters* (SC) [11] (Listing 1.6).

To evaluate the effectiveness of hardening we consider a set of fault models commonly used to find vulnerabilities in implementations of encryption algorithms [7]: branch instruction faults (both unconditional $B$ and conditional $BC$) that aim to disrupt the control flow; *FLP* faults flipping instruction bits; *NOP* faults that skip an instruction; and *Z1B* and *Z1W* faults that set a single instruction byte or word (respectively) to zero.

We consider fault injection on a binary level, as it remains one of the main attack vectors in the IoT environment [8, 9]. For our experiments we developed Chaos Duck[1] – an automatic tool that given a binary file produces "*faulted*" binaries (one binary per fault injected), and evaluates the impact of each fault.

We compare five implementations of PRESENT hardened with techniques described above with a non-hardened one (*i.e. baseline*)[2]. 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 differentiate between normally terminated binaries and those that were interrupted by timeout. We measure the total number of faulted binaries and collect statistics on fault types and their success rate. The hardening efficiency is evaluated based on its ability to prevent sensitive data leaks in general and under a differential fault attack (DFA) [5]. Finally, we measure the average execution time across 10000 executions with randomly generated *key,plaintext* pairs, and record the size in bytes for all binaries.

## 3 Results

With respect to data leaks, none of the hardening techniques were able to prevent plaintext leakage (see Table 1), which is in line with our previous results [10]. *FD* and *DaP* techniques were more effective since they check final output values instead of intermediate values, e.g. loop counters.

Branch instruction faults were the most common cause of sensitive data leaks (see Table 2). Hence hardening techniques that add more branches to the original code (e.g. *SC* technique) should be avoided, as new branches inadvertently increase the attack surface. Alternative techniques, e.g. *FD* and *DaP*, proved to be less affected by this type of fault.

[1] Available from `https://github.com/zavalyshyn/chaosduck`
[2] Available from `http://www.lightweightcrypto.org/implementations.php`

**Table 1.** Sensitive data leakage across five hardening techniques.

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Binaries** | **1314549** | **4818348** | **9267993** | **52341831** | **3580380** | **3902400** |
| Leaked key (normal/timeout) | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| Leaked plaintext (normal/timeout) | 1713 / 180 | 1449 / 0 | 3559 / 4 | 567 / 0 | 0 / 72 | 108 / 72 |
| DFA vulnerable | 2 / 0 | 9 / 0 | 2 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |

**Table 2.** Fault types statistics for faulted binaries leaking sensitive data (normally terminated vs. terminated by timeout).

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Binaries** | **1713 / 180** | **1449 / 0** | **3559 / 4** | **567 / 0** | **0 / 72** | **108 / 72** |
| FLP | 45 / 108 | 9 / 0 | 28 / 0 | 234 / 0 | 0 / 0 | 36 / 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 | 852 / 0 | 396 / 0 | 341 / 1 | 324 / 0 | 0 / 0 | 0 / 0 |
| BC | 816 / 72 | 1044 / 0 | 3190 / 3 | 0 / 0 | 0 / 72 | 72 / 72 |

**Table 3.** Execution results for five hardening techniques in presence of faults.

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Binaries** | **1314549** | **3836889** | **5639832** | **52341831** | **3580380** | **3902400** |
| Valid cipher | 16.7 % | 38.64 % | 34.07 % | 62.87 % | 16.99 % | 14.02 % |
| Invalid cipher | 17.3 % | 2 % | 1.64 % | 0.19 % | 0.57 % | 0.55 % |
| No output | 66 % | 59.36 % | 64.29 % | 36.94 % | 82.45 % | 85.43 % |

**Table 4.** Statistics on failed executions.

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Crashed binaries** | **867852** | **1862129** | **4807851** | **19334285** | **3505885** | **3845470** |
| Seg. fault | 56.11 % | 35.21 % | 20.99 % | 17.36 % | 45.28 % | 60.81 % |
| Timed out | 30.65 % | 6.81 % | 43.46 % | 0.48 % | 15.33 % | 9.87 % |
| Illegal instr. | 8.83 % | 1.82 % | 0.85 % | 0.65 % | 1.83 % | 2.25 % |
| Aborted | 0.63 % | 1.7 % | 0.68 % | 0.11 % | 0.29 % | 0.29 % |
| Fault detected | n/a | 52.4 % | 33.2 % | 80.22 % | 35.3 % | 25.03 % |
| Other | 3.78 % | 2.06 % | 0.82 % | 1.18 % | 1.97 % | 1.75 % |

Our experiments showed that the majority of faulted binaries simply failed to execute correctly without producing a valid ciphertext (see Table 3). However, the *SC* technique turned out to be less affected by the fault presence.

Failures were mostly caused by the binaries getting stuck in an infinite loop and then interrupted by timeout (see Table 4). The next most common cause was a segmentation fault, followed by illegal instruction and aborted faults. The fault detection rate is rather low, barely reaching 10% for most of the hardening techniques, with the sole exception of *CLH* reaching 80% detection rate.

All hardening techniques have no significant impact on execution time (52.4± 2 ms), and (except *SC*) have little impact on binary size (+1KB).

## 4 Conclusions

Hardening techniques exploring redundancy on a function level strike a good balance between security and performance properties, while some of the classic techniques make the software more vulnerable to faults resulting in a plaintext being leaked. Although the experiments were performed on PRESENT, these results indicate hardening is generally ineffective in preventing faults. Chaos Duck proved to be a powerful tool for evaluating an IoT software security and robustness in the presence of faults.

## References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. of IEEE **94**(2) (2006)
2. Barbu, G., Andouard, P., Giraud, C.: Dynamic fault injection countermeasure. In: Proc. of CARDIS (2012)
3. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: Present: An ultra-lightweight block cipher. In: Proc. of CHES (2007)
4. Cheynet, P., Nicolescu, B., Velazco, R., Rebaudengo, M., Reorda, M.S., Violante, M.: Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE TNS **47**(6) (2000)
5. Ghalaty, N.F., Yuce, B., Schaumont, P.: Differential fault intensity analysis on present and led block ciphers. In: Proc. of COSADE (2015)
6. Gilley, G.C., Bearnson, L., Carroll, C., Bouricius, W., Hsieh, E., Putzolu, G., Roth, J., Schneider, P., Tan, C., Hsiao, M., et al.: Ftcs, digest of papers (1971)
7. Given-Wilson, T., Heuser, A., Jafri, N., Legay, A.: An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. Concurr. Comput. Pract. Exp. **31**(23) (2019)
8. Given-Wilson, T., Jafri, N., Legay, A.: The state of fault injection vulnerability detection. In: Proc. of VECoS (2018)
9. Given-Wilson, T., Jafri, N., Legay, A.: Combined software and hardware fault injection vulnerability detection. ISSE (2020)
10. Given-Wilson, T., Legay, A.: Formalising fault injection and countermeasures. In: Proc. of ARES (2020)
11. Lalande, J.F., Heydemann, K., Berthomé, P.: Software countermeasures for control flow integrity of smart card c codes. In: Proc. of ESORICS (2020)