FISSC: a Fault Injection and Simulation Secure Collection

Louis Dureuil[1,2,3], Guillaume Petiot[1,3], Marie-Laure Potet[1,3], Thanh-Ha Le[4], Aude Crohen[4], and Philippe de Choudens[1,2]

[1] Univ. Grenoble Alpes, F-38000 Grenoble, France.
[2] CEA, LETI, MINATEC Campus, F-38054 Grenoble, France.
{louis.dureuil,philippe.de.choudens,cecile.dumas,jessy.clediere}@cea.fr
[3] CNRS, VERIMAG, F-38000 Grenoble, France.
{louis.dureuil,marie-laure.potet}@imag.fr
[4] Safran Morpho
{thanh-ha.le,aude.crohen}@morpho.com

Abstract. Applications in secure components (such as smartcards, mobile phones or secure dongles) must be hardened against fault injection to guarantee security even in the presence of a malicious fault. Crafting applications robust against fault injection is an open problem for all actors of the secure application development life cycle, which prompted the development of many simulation tools. A major difficulty for these tools is the absence of representative codes, criteria and metrics to evaluate or compare obtained results. We present FISSC, the first public code collection dedicated to the analysis of code robustness against fault injection attacks. FISSC provides a framework of various robust code implementations and an approach for comparing tools based on predefined attack scenarios.

1 Introduction

1.1 Security Assessment against Fault Injection Attacks

In 1997, Differential Fault Analysis (DFA) [6] demonstrated that unprotected cryptographic implementations are insecure against malicious fault injection, which is performed using specialized equipment such as a glitch generator, focused light (laser) or an electromagnetic injector [3]. Although fault attacks initially focused on cryptography, recent attacks target non-cryptographic properties of codes, such as modifying the control flow to skip security tests [16] or creating type confusion on Java cards in order to execute a malicious code [2].

Fault injections are modeled using various fault models, such as instruction skip [1], instruction replacement [10] or bitwise and byte-wise memory and register corruptions [6]. Fault models operate either at high-level (HL) on the source code or at low-level (LL) on the assembly or even the binary code. Both kinds of models are useful. HL models allow to perform faster and understandable analyses supplying a direct feedback about potential vulnerabilities. LL models

allow more accurate evaluations, as the results of fault injection directly depend on the compilation process and on the encoding of the binary.

Initially restricted to the domain of smartcards, fault attacks are nowadays taken into account in larger classes of secure components. For example the Protection Profile dedicated to *Trusted Execution Environment*[5] explicitly includes hardware attack paths such as power glitch fault injection. In the near future, developers of *Internet of Things* devices will use off-the-shelf components to build their systems, and will need means to protect them against fault attacks [8].

1.2 The Need for a Code Collection

In order to assist both the development and certification processes, several tools have been developed, either to analyze the robustness of applications against fault injection [10,5,4,14,8,7,11,13], or to harden applications by adding software countermeasures [15,9,12]. All these tools are dedicated to particular fault models and code levels. The main difficulty for these tools is the absence of representative and public codes allowing to evaluate and compare the relevance of their results. Partners of this paper are in this situation and have developed specific tools adapted to their needs: Lazart [14] an academic tool targeting multiple fault injection, Efs [4] an embedded LL simulator dedicated to developers and Celtic [7] tailored for evaluators.

In this paper, we describe FISSC (Fault Injection and Simulation Secure Collection), the first public collection dedicated to the analysis of secure codes against fault injection. We intend to provide (1) a set of representative applications associated with predefined attack scenarios, (2) an inventory of classic and published countermeasures and programming practices embedded into a set of implementations, and (3) a methodology for the analysis and comparison of results of various tools involving different fault models and code levels.

In Sec. 2, we explain how high-level attack scenarios are produced through an example. We then present the organization and the content of this collection in Sec. 3. Lastly in Sec. 4, we propose an approach for comparing attacks found on several tools, illustrated with results obtained from Celtic.

2 The VerifyPIN Example

Fig. 1 gives an implementation of a VerifyPIN command, allowing to compare a user PIN to the card PIN under the control of a number of tries. The `byteArrayCompare` function implements the comparison of PINs. Both functions illustrate some classic countermeasures and programming features. For example the constants `BOOL_TRUE` and `BOOL_FALSE` encode booleans with values more robust than `0` and `1` that are very sensible to data fault injection. The loop of `byteArrayCompare` is in fixed time, in order to prevent timing attacks. Finally, to detect fault injection consisting in skipping comparison, a countermeasure

[5] TEE Protection Profile. Tech. Rep. GPD_SPE_021. GlobalPlatform, november 2014

checks whether `i` is equal to `size` after the loop. The `countermeasure` function raises the global flag `g_countermeasure` and returns.

```
1  BOOL VerifyPIN() {
2    g_authenticated = BOOL_FALSE;
3    if(g_ptc > 0) {
4      if(byteArrayCompare(g_userPin,
5          g_cardPin, PIN_SIZE)
6            == BOOL_TRUE) {
7        g_ptc = 3;
8        g_authenticated = BOOL_TRUE;
9        return BOOL_TRUE;
10     } else {
11       g_ptc--;
12       return BOOL_FALSE;
13     }
14   } return BOOL_FALSE; }
```

```
15 BOOL byteArrayCompare(UBYTE* a1,
16     UBYTE* a2, UBYTE size) {
17   int i;
18   BOOL status = BOOL_FALSE;
19   BOOL diff = BOOL_FALSE;
20   for(i = 0; i < size; i++) {
21     if(a1[i] != a2[i]) {
22       diff = BOOL_TRUE; } }
23   if(i != size) {
24     countermeasure(); }
25   if(diff == BOOL_FALSE) {
26     status = BOOL_TRUE;
27   } else { status = BOOL_FALSE;
28   } return status; }
```

Fig. 1: Implementation of functions `VerifyPIN` and `byteArrayCompare`

To obtain high-level attack scenarios, we use the LAZART tool [14] which analyses the robustness of a source code (C-LLVM) against multiple control-flow fault injections (other types of faults can also be taken into account). The advantage of this approach is twofold: first, LAZART is based on a symbolic execution engine ensuring the coverage of all possible paths resulting from the chosen fault model; second, multiple injections encompass attacks that can be implemented as a single one in other fault models or low-level codes. Thus, according to the considered fault model, we obtain a set of significant high-level coarse-grained attack scenarios that can be easily understood by developers.

We apply LAZART to the VerifyPIN example to detect attacks where an attacker can authenticate itself with an invalid PIN without triggering a countermeasure. Successful attacks are detected with an oracle, i.e., a boolean condition on the C variables. Here: `g_countermeasure != 1 && g_authenticated == BOOL_TRUE`. We chose each byte of the user PIN distinct from its reference counterpart. Table 1 summarizes, for each vulnerability, the number of required faults, the targeted lines in the C code, and the effect of the faults on the application.

Number of faults	Fault injection locations	Effects
1	l. 25	invert the result of the condition
1	l. 4	invert the result of the condition
2	l. 20	do not execute the loop
	l. 23	do not trigger the countermeasure
4	l. 21 (four times)	invert each byte check

Table 1: High-level attacks found by LAZART and their effects

In FISSC, for each attack, we provide a file containing the chosen inputs and fault injection locations (in terms of basic blocks of the control flow graph) as well as a colored graph indicating how the control flow has been modified. Detailed results for this example can be found on the website.[6]

[6] `http://sertif-projet.forge.imag.fr/documents/VerifyPIN_2_results.pdf`

3 The FISSC Framework

As pointed out before, FISSC targets tools working at various code levels and high-level attack scenarios can be used as reference to interpret low-level attacks. Then, we supply codes at various levels and the preconized approach is described in Fig. 2 and illustrated in Sec. 4.

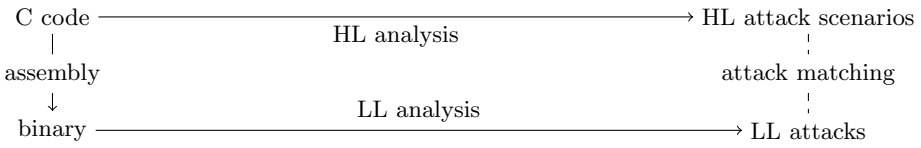

Fig. 2: Matching LL attacks with HL attack scenarios

In this current configuration, FISSC supports the C language and the ARMv7 M (Cortex M4) assembly. We do not distribute binaries targeting a specific device, but they can be generated by completing the gcc linker scripts.

3.1 Contents and File Organization

The first release of FISSC contains small basic functions of cryptographic implementations (key copy, generation of random number, RSA) and a suite of VerifyPIN implementations of various robustness, detailed in section 3.2. For these examples, Table 2 describes oracles determining attacks that are considered successful. For instance attacks against the VerifyPIN command target either to be authenticated with a wrong PIN or to get as many tries as wanted. Attacks against AESAddRoundKeyCopy try to assign a known value to the key in order to make the encryption algorithm deterministic. Attacks against GetChallenge try to prevent the random buffer generation, so that the challenge buffer is left unchanged. Attacks against CRT-RSA target the signature computation, so that the attacker can retrieve a prime factor p or q of N.

Example	Oracle
VerifyPIN	`g_authenticated == 1`
VerifyPIN	`g_ptc >= 3`
AES KeyCopy	`g_key[0] = g_expect[0] || ... || g_key[N-1] = g_expect[N-1]`
GetChallenge	`g_challenge == g_previousChallenge`
CRT-RSA	`(g_cp == pow(m,dp)% p && g_cq != pow(m,dq)% q)` `|| (g_cp != pow(m,dp)% p && g_cq == pow(m,dq)% q)`

Table 2: Oracles in FISSC

Each example is split into several C files, with a file containing the actual code, and other files providing the necessary environment (e.g., countermeasure, oracle, initialization) as well as an interface to embed the code on a device (types, NVM memory read/write functions). This modularity allows one to use the implementation while replacing parts of the analysis or interface environments.

3.2 The VerifyPIN Suite

Applications are hardened against fault injections by means of countermeasures (CM) and programming features (PF). Countermeasures denote specific code designed to detect abnormal behaviors. Programming Features denote implementation choices impacting fault injection sensitivity. For instance, introducing function calls or inlining them introduces instructions to pass parameters, which changes the attack surface for fault injections. Table 4 lists a subset of classic and published PF and CM we are taking into account. The objective of the suite is not to provide a fully robust implementation, but to observe the effect of the implemented CM and PF on the produced attack scenarios.

	HB	FTL	INL	BK	SC	DT	# scenarios for i faults				
							1	2	3	4	Σ
v0							2	0	0	1	3
v1	✓						2	0	0	1	3
v2	✓	✓			✓		2	1	0	1	4
v3	✓	✓	✓		✓		2	1	0	1	4
v4	✓	✓	✓	✓	✓		2	0	1	1	4
v5	✓	✓			✓	✓	0	4	4	1	9
v6	✓	✓	✓			✓	0	3	0	1	4
v7	✓	✓	✓		✓	✓	0	2	0	0	2

Table 3: PF/CM embedded in VerifyPIN suite

PF	
INL	Inlined calls
FTL	Fixed time loop
CM	
HB	Hardened booleans
BK	Backup copy
DT	Double test
SC	Step counter

Table 4: List of CM/PF

Table 3 gives the distribution of CM and PF in each implementation (v2 is the example of Fig. 1). Hardened booleans protect against faults modifying data-bytes. Fixed-time loops protect against temporal side-channel attacks. Step counters check the number of loop iterations. Inlining the `byteArrayCompare` function protects against faults changing the call to a NOP. Backup copy prevents against 1-fault attacks targeting the data. Double call to `byteArrayCompare` and double tests prevent single fault attacks, which become double fault attacks. Calling a function twice (v5) doubles the attack surface on this function. Step counters protect against all attacks disrupting the control flow integrity [9].

4 Comparing Tools

The HL scenarios and oracles defined in Sec. 2-3 allow for the comparison of tools in the FISCC framework. In particular, the successful attacks discovered by tools should cover the HL scenarios. In order to associate HL scenarios and attacks we propose several *Attack Matching* criteria. *Attack matching* consists in deciding whether some attacks found by a tool are related to attacks found by another tool. An attack is *unmatched* if it is not related to any other attack.

In [5], HL faults are compared with LL faults with the following criterion: attacks that lead to the same program output are considered as matching. This "functional" criterion is not always discriminating enough. For instance, codes

like verifyPIN produce a very limited set of possible outputs ("authenticated" or not). We propose two additional criteria:
Matching by address. Match attacks that target the same address. To match LL and HL attacks, one must additionally locate the C address corresponding to the assembly address of the LL attack.
Fault Model Matching. Interpret faults in one fault model as faults in the other fault model. For instance, since conditional HL statements are usually compiled to `cmp` and `jmp` instructions, it makes sense to interpret corruptions of `cmp` or `jmp` instructions (in the *instruction replacement* fault model) as test inversions.

4.1 Case Study

We apply our criteria to compare the results of CELTIC and LAZART on the example of Fig. 1. In our experiments, CELTIC uses the *instruction replacement* fault model, where a single byte of the code is replaced by another value at runtime. Testing the possible values exhaustively, CELTIC finds 432 successful attacks. We then apply our two matching criteria to these results. Fig. 3 indicates the number of successful attacks per address of assembly code, and the (manually determined) correspondence between assembly addresses and C lines. The C lines *4*, *20*, *21*, *23* and *25* correspond to the scenarios found by LAZART in Table 1. They are matched *by address* with the attacks found by CELTIC. CELTIC attacks that target a `jump` or a `compare` instruction are also matched *by fault model*.

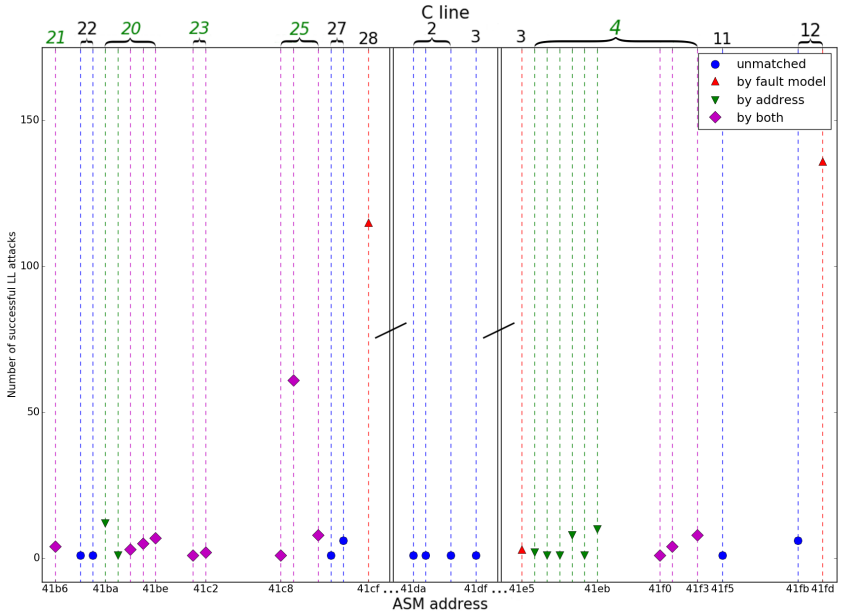

Fig. 3: Matching HL and LL attacks

4.2 Interpretation

Fault model matching can be used to quickly identify HL-attacks amongst LL-attacks with only a hint of the correspondence between C and assembly, while address matching allows to precisely find the HL-attacks matched by the LL-attacks. Both matching criteria yield complementary results. For instance, attacks at address 0x41eb are matched only by address, while attacks at 0x41fd only by fault model.

Interestingly, some multiple fault scenarios of Lazart are implemented by single fault attacks in Celtic. For instance, the 4-fault scenario of l.21 is implemented with the attacks at address 0x41b6. In the HL scenario the conditional test inside the loop is inverted 4 consecutive times. In the LL attacks, The corresponding jump instruction is actually not inverted, but its target is replaced so that it jumps to l.26 instead of l.22. These attacks are matched with our current criteria, although they are semantically very different. Lastly, 20 LL-attacks remain unmatched. They are subtle attacks that depend on the encoding of the binary or on a very specific byte being injected. For instance, at 0x41da, the value for `BOOL_FALSE` is replaced by the value for `BOOL_TRUE`. This is likely to be hard to achieve with actual attack equipment.

In this example, *attack matching* criteria allows to show that Celtic attacks cover each HL-scenario. Other tools can use this approach to compare their results with those of Celtic and the HL-scenario of Lazart. Their results should cover the HL-scenario, or offer explanations (for instance, due to the fault model) if the coverage is not complete.

5 Conclusion

FISSC is available on request.[7] It can be used by tool developers to evaluate their implementation against many fault models and it can be contributed to with new countermeasures (the first external contribution is the countermeasure of [9]). We plan to add more examples in the future releases of FISSC (e.g. hardened DES implementations) and to extend Lazart to simulate faults on data.

Acknowledgments. This work has been partially supported by the SERTIF project (ANR-14-ASTR-0003-01): `http://sertif-projet.forge.imag.fr` and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025).

References

1. Anderson, R., Kuhn, M.: Low cost attacks on tamper resistant devices. In: Christianson, B., Crispo, B., Lomas, M., Roe, M. (eds.) Security Protocols, LNCS, vol. 1361, pp. 125–136. Springer (1998)

[7] To request or contribute, send an e-mail to `sertif-secure-collection@imag.fr`.

2. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) Smart Card Research and Advanced Application. 9th IFIP WG 8.8/11.2 International Conference. LNCS/Security & Cryptology, vol. 6035, pp. 148–163. Springer (2010)
3. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Pr. of the IEEE 100(11), 3056–3076 (2012)
4. Berthier, M., Bringer, J., Chabanne, H., Le, T.H., Rivière, L., Servant, V.: Idea: Embedded fault injection simulator on smartcard. In: ESSoS. LNCS, vol. 8364, pp. 222–229. Springer (2014)
5. Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J.: High Level Model of Control Flow Attacks for Smart Card Functional Security. In: ARES 2012. pp. 224–229. IEEE (2012)
6. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Advances in Cryptology–EUROCRYPT97. pp. 37–51. Springer (1997)
7. Dureuil, L., Potet, M.L., Choudens, P.d., Dumas, C., Clédière, J.: From code review to fault injection attacks: Filling the gap using fault model inference. In: 14th Smart Card Research and Advanced Application Conference, CARDIS15. LNCS (2015)
8. Holler, A., Krieg, A., Rauter, T., Iber, J., Kreiner, C.: Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks. In: Digital System Design (DSD), Euromicro15. pp. 530–533. IEEE (2015)
9. Lalande, J., Heydemann, K., Berthomé, P.: Software countermeasures for control flow integrity of smart card C codes. In: Pr. of the 19th European Symposium on Research in Computer Security, ESORICS 2014. pp. 200–218 (2014)
10. Machemie, J.B., Mazin, C., Lanet, J.L., Cartigny, J.: SmartCM a smart card fault injection simulator. In: IEEE International Workshop on Information Forensics and Security. IEEE (2011)
11. Meola, M.L., Walker, D.: Faulty logic: Reasoning about fault tolerant programs. In: 19th European Symposium on Programming, ESOP. pp. 468–487. Springer (2010)
12. Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks. J. Cryptographic Engineering 4(3), 145–156 (2014)
13. Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: Discovering application-level insider attacks using symbolic execution. In: 24th IFIP TC 11 International Information Security Conference, SEC 2009. pp. 63–75. Springer (2009)
14. Potet, M.L., Mounier, L., Puys, M., Dureuil, L.: Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014. pp. 213–222. IEEE (2014)
15. Séré, A., Lanet, J.L., Iguchi-Cartigny, J.: Evaluation of countermeasures against fault attacks on smart cards. International Journal of Security and Its Applications 5(2) (2011)
16. Van Woudenberg, J.G., Witteman, M.F., Menarini, F.: Practical optical fault injection on secure microcontrollers. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on. pp. 91–99. IEEE (2011)