Comprehensive Analysis of Software Countermeasures Against Fault Attacks

Nikolaus Theißing*, Dominik Merli†, Michael Smola‡, Frederic Stumpf†, Georg Sigl§
*Institute of Flight Systems, University of the Armed Forces, Munich, Germany
Email: nikolaus.theissing@unibw.de
†Fraunhofer Research Institution for Applied and Integrated Security (AISEC), Munich, Germany
Email: {dominik.merli, frederic.stumpf}@aisec.fraunhofer.de
‡Infineon Technologies AG, Munich, Germany
Email: michael.smola@infineon.com
§Institute for Security in Information Technology, Technische Universität München, Munich, Germany
Email: sigl@tum.de

***Abstract*—Fault tolerant software against fault attacks constitutes an important class of countermeasures for embedded systems. In this work, we implemented and systematically analyzed a comprehensive set of 19 different strategies for software countermeasures with respect to protection effectiveness as well as time and memory efficiency. We evaluated the performance and security of all implementations by fault injections into a microcontroller simulator based on an ARM Cortex-M3. Our results show that some rather simple countermeasures outperform other more sophisticated methods due to their low memory and/or performance overhead. Further, combinations of countermeasures show strong characteristics and can lead to a high fault coverage, while keeping additional resources at a minimum. The results obtained in this study provide developers of secure software for embedded systems with a solid basis to decide on the right type of fault attack countermeasure for their application.**

I. Introduction

In 1997, Boneh et al. [1] were able to conduct an attack on a cryptographic microcontroller with a method which was later to be known as a *fault attack*. Unlike previous cryptanalysts, who had attempted to break cryptosystems purely by analyzing mathematical weaknesses inside their algorithms, Boneh et al. purposely injected faults into the circuit of the actual implementation. These faults caused the implemented cipher, in this case RSA [2], to behave erroneously. The induced errors made it possible for the researchers to break the cryptographic key in polynomial time.

On the practical side, it was shown that attackers have a wide range of possibilities to cause faults within integrated circuits. They might vary operating conditions, e.g. by heating the circuit [3] or manipulating the power supply [4], [5] by inducing glitches. Also, radiation of all kinds enables fault injection, e.g. cosmic radiation [3], X-ray [6], light [7] and laser beams [8]. Invasive methods, like microprobing [9] allow injecting faults precisely at targeted signals.

Physical faults can cause three different types of errors in software. They can (1) directly affect critical data values, (2) cause erroneous calculations or (3) lead to Control-Flow Errors (CFE) by influencing the instruction sequence of a program. To handle these different types of errors, several countermeasures [10]–[14] have been proposed. However, software developers cannot focus on countermeasure development and evaluation, but need a supportive analysis of effectiveness and performance overhead of protection ideas as a basis to decide on the most suitable countermeasure(s) to use.

Our contribution is a comprehensive analysis of software-implemented countermeasures against fault attacks. We simulated a variety of countermeasures while injecting random bit faults into registers and memory of a microcontroller simulator. Our results give an insight into effectiveness and performance of these protective software modules and are therefore a strong foundation for designers of secure embedded software.

The rest of this paper is organized as follows. Section II gives an overview of publications on related work. Section III presents our simulation system, benchmark application and assessment approach. In Section IV, the list of implemented and tested software countermeasures is presented. Section V provides data and interpretation of the observed experimental results, followed by a conclusion in Section VI.

II. Related Work

Since the exploitation of fault attacks by Boneh et al. [1], their main focus has been breaking cryptographic protocols. A lot of research has been conducted on methods protecting implementations of cryptographic protocols against fault attacks, e.g. the protection of RSA [2] by Blömer et al. [15], Kim and Quisquater [16], and Bao et al. [17]. Countermeasures hardening implementations of Schnorr and Fiat-Shamir authentication were presented by Boneh et al. [1] and a protected version of the El-Gamal key exchange protocol was shown by Bao et al. [17]. In our simulation setup, we also use an authentication application to evaluate each countermeasure.

978-3-9815370-0-0/DATE13/©2013 EDAA

III. Methodology

In this section, we give an overview of the evaluation system we used to inject faults into a program running on a microcontroller simulator. Further, we explain the authentication benchmark software which was used for testing.

A. Fault Simulation

In order to be able to simulate a large amount of fault injections for several different software versions on a state-of-the-art microcontroller, we used a simulator of an ARM Cortex-M3. As shown in Figure 1, we implemented an injection mechanism which allowed us to stop the simulation at every given point in time in order to inspect or modify the system state, namely the contents of all registers of the microcontroller.

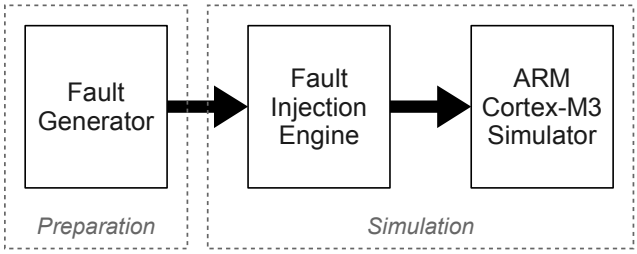

Fig. 1. Fault simulation system

We further implemented a fault generator, which was able to generate a list of faults covering the whole simulation space created by time $\times$ memory elements. We used the single-bit fault model [1] to created transient faults randomly affecting any register or memory cell at a random point in time. This is a very suitable and practical definition of a limited attacker, who is able to manipulate single bits with little or no precision. Since extensive simulations require an extreme amount of time and computational resources, we added a feature to reduce the number of necessary simulations, i.e. it was possible to reduce the number of memory addresses or the number of single-bit faults per data word, while maintaining a uniform distribution over the simulation space.

B. Benchmark Application

We implemented an operating system for a microcontroller within a fictitious money card to simulate it on our Cortex-M3 simulator. This card works analogously to existing pay cards [18]. After authentication by the user's PIN, one can charge money to the card and pay with it at payment stations. Additionally, a mutual challenge-response authentication protocol based on the Advanced Encryption Standard (AES) takes place between the card and the charging/payment stations.

The software consists of three separate program parts:

- **Module A:** Mutual authentication based on AES
- **Module B:** PIN verification
- **Module C:** Balance transaction

The AES key management, encryption and decryption was purely carried out in software. However, this software part was spared from fault simulations throughout this work, since in real security microcontrollers, the AES will be computed by specific hardware circuits protected by hardware countermeasures [19].

C. Simulation Runs and Assessment

In order to assess the result state of a simulation run, we separated the simulation space into six distinct subsets, i.e., every module was simulated once for register faults and RAM faults, respectively. At each simulation run, the whole program was executed, but faults were only injected into the module-under-test.

For module A, the software was provided with invalid authentication data, i.e., a critical fault would lead to positive authentication without valid data. While testing module B, only a wrong PIN was supplied. In module C, all data is correctly input, thus, a critical error is only recognized if the balance value is manipulated during a transfer.

Following the definitions above, we obtained one of the following results after each simulation run:

- **Pass.** The fault has not been security-critical, or in case of a critical fault, the processor has entered its error state, i.e., the fault was detected.
- **Fail.** The normal terminal state could be reached without valid authentication data or PIN, or the balance value was modified.
- **Timeout.** The microcontroller continues without ever or at least very late reaching the terminal or the error state. Reasons might be CFEs causing infinite or long-lasting loops or unhandled interrupts causing a system halt. As in hardware, these cases are detected by a timer and can be assessed as successful fault detections.

IV. Software Countermeasures

This section describes all implemented countermeasures. Each test scenario is assigned a two-digit code number where the first one represents the protection classes listed below and the second one is a sequence number. Countermeasure combinations were chosen in a way that they compensate each other's weaknesses.

- **00**: Original software without countermeasures
- **1x**: Countermeasures protecting the data layer
- **2x**: Combinations of data protection methods
- **3x**: Countermeasures protecting the control flow layer
- **4x**: Combinations of control flow protection methods
- **5x**: Combinations of data and control flow protection

Following countermeasures were implemented and simulated:

- **Test 10: Instruction Duplication:**
  A calculation is carried out multiple times (here: twice) and the result is assigned to two variables. Afterwards, both results are checked for equality.

- **Test 11: Data Duplication:**
  Redundant copies of all data elements are created [10] and operations are performed for every data copy. Comparing all copies of the respective element, one is able to detect data errors at an arbitrary point in time.
- **Test 12: Inverse Operations:**
  Application of the inverse operation on the operation result of a preceding calculation and assertion that the result is equal to the initial parameter of the first calculation.
- **Test 13: Operand Modification:**
  Calculations are carried out multiple times (here: twice). The first calculation is carried out with the original values of the operands. Then, the operands are modified in a certain way, the calculation is performed again and the modification is inverted before comparison [11].
- **Test 14: Acceptance Tests:**
  Acceptance Tests make use of a-priori information of the programmer. They check for a valid operation result by asserting that the respective value lies within a known subset of valid assignments [12].
- **Test 15: Error-Detecting Parity Codes:**
  Alongside the data elements to be stored, a parity data word, i.e. an XOR of all data elements, is saved. Later, this redundant data is used to verify the integrity of the stored information [20], [21].
- **Test 20: Data Duplication + Operand Modification:**
  A combination of the two methods of tests 11 and 13.
- **Test 21: Inverse Operations + Operand Modification:**
  A combination of the two methods of tests 12 and 13.
- **Test 30: Jump ID:**
  Before a branch to a remote code block takes place, the ID of the target block is assigned to a signature variable which can be verified by every block. An error is detected if the target ID and the block ID do not match [22].
- **Test 31: Jump ID (Late Check):**
  The same principle as in test 30 is applied, but the ID checking step is conducted at the end of each block instead of at the beginning.
- **Test 32: Source and Destination ID:**
  Similarly to the concept of Jump ID, a signature of the target block as well as one of the originating block is stored. The target block is then able to verify the transition between these blocks.
- **Test 33: Control-Flow Checking by Software Signatures (CFCSS):**
  A dynamic signature variable is updated before each branch according to the IDs of both blocks. Again, the target block verifies the transition [13], [14].
- **Test 34: Assertions for Control Flow Checking (ACFC):**
  ACFC utilizes a global variable to monitor the execution flow. It keeps track of a part of the history of the recently executed blocks and assures that the protected program is valid according to a predefined control flow graph [23].
- **Test 35: Incremental Checkpassing ID:**
  Checkpoints are placed at various positions within a block. Whenever the program execution reaches a checkpoint, a variable is manipulated in a certain way. At the end of the block, a check is executed which ascertains that each checkpoint was processed [10].
- **Test 40: Source-Destination-ID + Checkpassing:**
  A combination of the two methods of tests 32 and 35.
- **Test 41: Source-Destination-ID + Condition Checks:**
  A combination of the method described in test 32 and condition checks, which compute conditional expressions redundantly to assure that a conditional branch is valid.
- **Test 50: Jump ID + Data Duplication:**
  A combination of the data-protecting method described in test 11 and the control-flow-protecting method described in test 30.
- **Test 51: Source-Destination-ID + Checkpassing + Inverse Operations + Operand Modification:**
  A combination of the data-protecting methods described in tests 12 and 13, and the control-flow-protecting methods described in tests 32 and 35.

V. Result Analysis

This section discusses the results of our experiments regarding critical errors, fault coverage and performance. For each countermeasure, depending on the implementation, simulations covering faults in all registers at all times required $20,000$ to $180,000$ simulation runs, $85,000$ on average. For RAM faults in a randomly selected part of the used RAM at all times, a number of $150,000$ to $280,000$ simulation runs were performed, 235.000 on average per countermeasure.

A. Critical Errors

The results of the fault simulation on the original software (test 00) allow for a qualitative analysis of critical errors caused by injected faults and a quantitative analysis of their distribution.

In modules A and B, the injection of register faults led to the following distinct types of critical errors:

- Data errors in parts of the authentication data
- Data errors in branching condition variables
- Data errors in the value of the Link Register, resulting in a stack storage of an invalid return address
- Intra-block CFEs skipping a `PUSH LR`[1] instruction
- Inter-block CFEs jumping into wrong conditional blocks
- Inter-block CFEs jumping into other routines

Our analysis showed that the vast majority of errors is comprised of inter-block CFEs jumping to remote routines. At the beginning of the routine querying the PIN or conducting the authentication, the return address (stored in the Link Register) is pushed onto the stack. It points to the instruction in the main routine where execution should be continued after a successful authentication or PIN query. After the inter-block CFE, the code of the remote block is executed. At the end

[1] store Link Register (LR) to stack

of that block, the return address is restored to the Instruction Pointer via a `POP IP`[2] instruction. At that point, because the initial `PUSH LR` of the remote block has been skipped, the invalid former return address is loaded into the Instruction Pointer. Then, the code execution continues at the position immediately after the successful authentication or PIN query.

Register faults during the execution of module C caused these critical errors:

- Intra-block CFEs skipping instructions on critical data
- Data errors modifying the stack pointer
- Data errors in branching condition variables
- Data errors in critical data

There, according to our analysis, the majority of errors is equally shared by two types of data errors. One half is comprised of data errors in the critical data itself, i.e. the value of the current balance or the booking difference. The other data errors affect elements responsible for conditional jumps. The control flow in module C consists of a switch-case construct, where faults in the switching variable are causing critical errors of this kind. The critical errors caused by memory faults in module C entirely consisted of modified critical data, i.e. of modified balance values.

B. Fault Coverage

To measure the fault coverage of an implemented countermeasure, we used the absolute number of failed tests. It yields the number of undetected faults within the used subset of the fault space. The smaller this number is, the higher is the error detection of the tested module.

1) Modules A and B, Register Faults: Modules A and B are mostly vulnerable to CFEs. The results in Figure 2 show that countermeasures only protecting the data layer (tests 10 to 21) provide only little (if any) coverage against faults causing security-critical errors.

The best results were achieved by test 41. The employed countermeasure combines block signatures with redundant checks of branching conditions.

As shown in Section V-A, a vast majority of critical CFEs is comprised of invalid inter-block CFEs according to the control-flow graph. Most of these errors can be detected by all CFE-detecting methods. A significant part of the remaining errors, however, is caused by invalid executions of conditional branches either by an erroneous jump or an erroneous evaluation of a jump condition. The methods implemented in test 41 are able to detect most of those CFEs and hence yield an exceptionally high CFE coverage.

Notably, the countermeasure from test 41 even outperforms the two highly sophisticated methods ACFC and CFCSS. Reasons therefore are, that on one hand, both countermeasures do not protect against data errors and graph-valid inter-block CFEs. On the other hand, their greater overhead provides an attacker a greater time interval to inject a fault.

2) Module C, Register Faults: Module C is mostly vulnerable against errors in the data layer, as shown in Section V-A. Some faults causing data errors immediately affect the critical data. The rest of the inflicted faults cause CFEs which indirectly affect the critical data. Even most of the CFEs are created by data errors at first hand.

Countermeasures protecting the control flow can hence provide protection to a certain degree. A more extensive protection, however, is given by data protection methods, as shown in Figure 3.

The data protection methods Data Duplication, Inverse Operations, and Operand Modification protect data elements not only stored in RAM, but also in the processor registers. Henceforth, in this scenario, countermeasures employing at least one of these methods provide a high fault coverage.

The two methods in tests 50 and 51, combining the mentioned data protection methods with additional protection against CFEs, provide the highest degree of protection. Especially, the two complementary methods combined in test 50 perform extremely well.

3) Module C, RAM Faults: For RAM faults, the measured number of undetected faults corresponds only to a reduced fault space, i.e. only a randomly selected part of the RAM addresses have been used, because of computational limitations. Hence, to achieve comparable numbers, the measured number of undetected faults is divided by the sample size and multiplied by the scaling normalization factor of $2 \cdot 10^5$

[2]fetch Instruction Pointer (IP) from stack

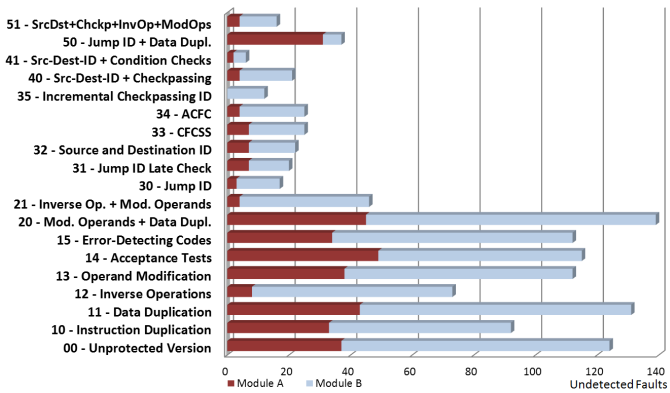

Fig. 2. Number of undetected register faults in modules A and B

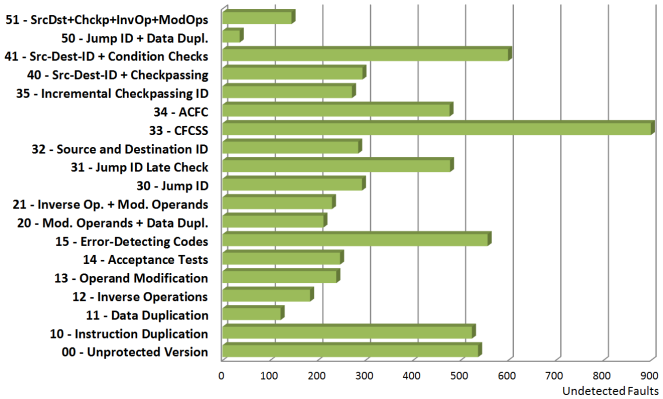

Fig. 3. Number of undetected register faults in module C

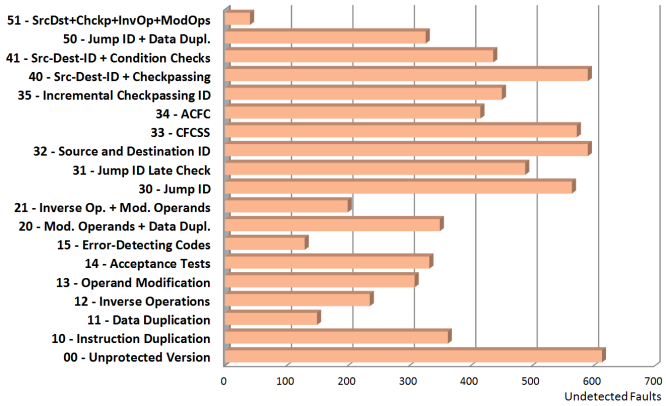

Fig. 4. Normalzied number of undetected RAM faults in module C

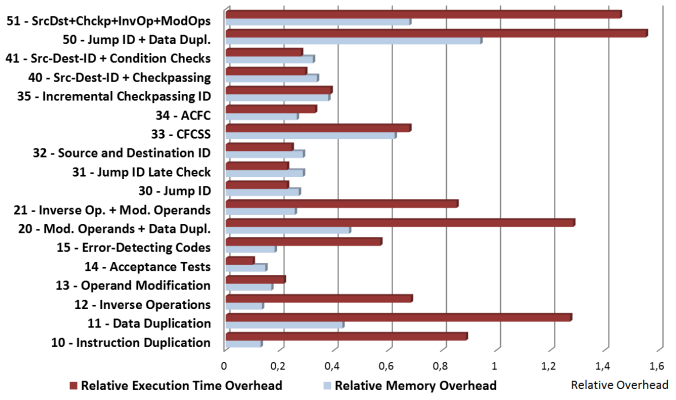

Fig. 6. Countermeasure overheads regarding execution time and memory consumption, relative to the values of the original program

faults, which has no deeper meaning. The resulting numbers are depicted in Figure 4.

Data Duplication yields a good fault coverage, but it is slightly outperformed by the countermeasure employing error-detecting codes. While these methods provide only little coverage against register faults affecting the control flow, they protect RAM data to a great extent. Test 51 achieves the greatest fault coverage in this scenario.

4) Overall Fault Coverage: Figure 5 depicts the overall relative number of failed tests, i.e. the sum of the number of undetected faults in all three modules for register as well as memory faults, divided by the sum of all injected faults.

The method of Data Duplication yields the best results when considering test sets with only one countermeasure strategy. Tests 50 and 51, combining data and control flow protection, show the lowest relative number of undetected faults and, henceforth, the best overall fault coverage.

C. Performance

This section discusses the performance penalty of all software countermeasures, as it is an important criterion for many applications.

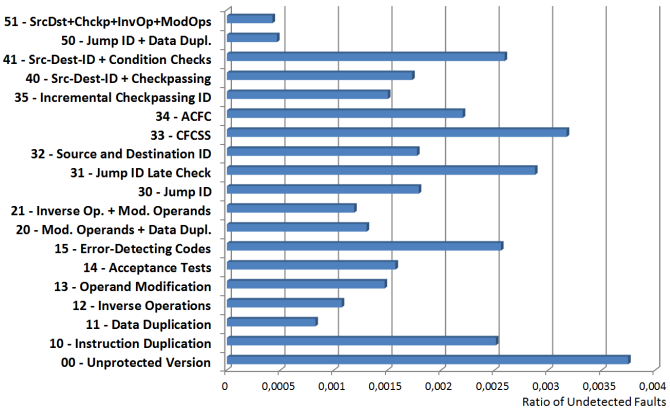

Fig. 5. Overall number of undetected faults, including all three modules

1) Memory Overhead: Figure 6 depicts the relative overhead of the countermeasures regarding their execution time and their overall consumed memory.

Test 50, combining Jump IDs and Data Duplication, consumes the highest portion of RAM. The overhead is significantly larger than the sum of the overheads of both countermeasures in their respective standalone implementations. The reason is that the combined countermeasure was created by first protecting the original software by Data Duplication and afterwards applying the Jump ID method on the protected, larger version. The standalone countermeasure with the largest memory overhead is CFCSS (test 33). Its low consumption of data memory implies that most of the created overhead is caused by the additional code executed by the method.

The lowest memory overhead is created by the countermeasures Instruction Duplication (test 10), Inverse Operations (test 12), and Acceptance Tests (test 14). These methods combine a low amount of additional code with little temporarily stored data.

2) Overhead in Execution Time: The highest overheads regarding execution time are created by the two combined methods from tests 50 and 51, followed by the combination of Data Duplication and Modified Operands (test 20) and the standalone implementation of Data Duplication (test 11). All of them utilize a high amount of redundant function calls.

The lowest temporal overhead for methods protecting the data layer is achieved by Acceptance Tests (test 14) since they get along with a low amount of redundant executions, followed by Operand Modification (test 13).

Out of the methods protecting the control flow, those consuming the lowest amount of execution time are those methods utilizing the least complicated checks. These are Jump ID methods (tests 30 and 31) as well as Source and Destination ID Check (test 32).

3) Coverage vs. Overhead: The combination of Source and Destination IDs with redundant Condition Checks (test 41) yields the best trade-off between coverage and overhead regarding CFEs with high distinction. It combines a below-average overhead with the best achieved fault coverage of all

performed tests.

The methods utilizing simple IDs for keeping track of the inter-block control flow (tests 30, 31, 32, and 35) also yield good compromises. They even outperform the sophisticated methods CFCSS (test 33) and ACFC (test 34), which yield a similar fault coverage, but impose a higher overhead.

The best balance of coverage and overhead for faults affecting data elements is achieved by Acceptance Tests (test 14), because of their low resource consumption. The second best methods are tests 50 and 51 for register and RAM faults, respectively, because of their very high coverage.

The combination of a fault coverage which is moderately worse than the one of Data Duplication and a significantly better resource consumption, makes the Operand Modification (Test 13) the method with the third best values.

VI. Conclusion

In this paper, we comprehensively analyzed software-implemented countermeasures against fault attacks covering methods protecting the data layer as well as the control flow. We described our simulation and evaluation setup and listed several state-of-the-art countermeasures with short explanations. The main contribution of this work are extensive fault simulations of our pay card benchmark application implemented in 19 different countermeasure scenarios. These tests yielded quantitative data about the effectiveness against fault attacks and the performance of the evaluated software countermeasures.

The best overall result was achieved by a combination of redundant condition checks and source and destination IDs, which reached the best fault coverage while implying a moderate performance overhead. Further, our analysis shows that simple countermeasures consisting of ID-based inter-block control flow checking are able to outperform sophisticated methods like CFCSS and ACFC.

References

[1] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *EUROCRYPT*, 1997, pp. 37–51.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, February 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[3] S. Govindavajhala and A. W. Appel, "Using memory errors to attack a virtual machine," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, ser. SP '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 154–. [Online]. Available: http://portal.acm.org/citation.cfm?id=829515.830563

[4] A. Barenghi, G. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, "Low voltage fault attacks to AES and RSA on general purpose processors," Cryptology ePrint Archive, Report 2010/130, 2010, http://eprint.iacr.org/.

[5] O. Kömmerling and M. Kuhn, "Design principles for tamper-resistant smartcard processors," in *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*. USENIX Association, 1999.

[6] M. Otto, "Fault attacks and countermeasures," Ph.D. dissertation, University of Paderborn, 2005. [Online]. Available: http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/DissertationMartinOtto.pdf

[7] V. Oklobdzija, *The Computer Engineering Handbook*. Boca Raton: CRC Press, 2008.

[8] M. G. Karpovsky, K. J. Kulikowski, and A. Taubin, "Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard," in *DSN*, 2004, pp. 93–101.

[9] S. P. Skorobogatov, "Semi-invasive attacks – A new approach to hardware security analysis," University of Cambridge, Tech. Rep. UCAM-CL-TR-630, 2005. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf

[10] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[11] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Trans. Comput.*, vol. 31, pp. 589–595, July 1982. [Online]. Available: http://portal.acm.org/citation.cfm?id=1311065.1311154

[12] M. Rela, H. Madeira, and J. Silva, "Experimental evaluation of the fail-silent behaviour in programs with consistency checks," *Fault-Tolerant Computing, International Symposium on*, vol. 0, p. 394, 1996.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE TRANSACTIONS ON RELIABILITY*, vol. 51, pp. 111–122, 2002.

[14] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *In Proceedings of the 3rd International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[15] J. Blömer, M. Otto, and J.-P. Seifert, "A new CRT-RSA algorithm secure against bellcore attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*. ACM Press, 2003, pp. 311–320. [Online]. Available: http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/SecureCRT.pdf

[16] C. H. Kim and J.-J. Quisquater, "How can we overcome both side channel analysis and fault attacks on RSA-CRT?" in *FDTC*, 2007, pp. 21–29.

[17] F. Bao, R. H. Deng, Y. Han, A. B. Jeng, A. D. Narasimhalu, and T.-H. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults," in *Proceedings of the 5th International Workshop on Security Protocols*. London, UK: Springer-Verlag, 1998, pp. 115–124. [Online]. Available: http://portal.acm.org/citation.cfm?id=647215.720385

[18] EMVCo, www.emvco.com.

[19] M. Joye and M. Tunstall, Eds., *Fault Analysis in Cryptography*. Springer Berlin Heidelberg, 2012. [Online]. Available: http://ebooks.ub.uni-muenchen.de/30229/

[20] K. J. Kulikowski, M. G. Karpovsky, and A. Taubin, "Fault attack resistant cryptographic hardware with uniform error detection," in *FDTC*, 2006, pp. 185–195.

[21] T. G. Malkin, F.-X. St, and M. Yung, "A comparative cost/security analysis of fault attack countermeasures," in *In Second Workshop on Fault Detection and Tolerance in Cryptography (FDTC 2005*, 2005, pp. 109–123.

[22] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip faults," *Nuclear Science, IEEE Transactions on*, vol. 51, no. 6, pp. 3510 – 3518, 12 2004.

[23] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," *On-Line Testing Symposium, IEEE International*, vol. 0, p. 137, 2003.