Fault-Channel Watermarks

Peter Samarin[1,2], Alexander Skripnik[1], and Kerstin Lemke-Rust[1]

[1] Bonn-Rhein-Sieg University of Applied Sciences, Germany
[2] Ruhr-Universität Bochum, Germany
peter.samarin@h-brs.de, alexander.skripnik@smail.inf.h-brs.de,
kerstin.lemke-rust@h-brs.de

Abstract. We introduce a new approach for securing intellectual property in embedded software implementations by using the response of an implementation to fault injections. In our approach, the implementation serves as its own watermark that is recorded through its fault effects. There is no additional code for the watermark. A simulator that maps the fault injections to the executed instructions aids an automated characterization of program code. We provide a proof-of-concept implementation of our watermarking approach using an 8-bit ATMega163 microcontroller and several assembly implementations of AES encryption. The results show that our method is well-suited for detection of identical software copies. In addition, our method shows robust performance in detection of software copies with a large number of added dummy instructions.

Keywords: Fault-Channel Watermarks, Watermarking, IP Protection, Fault Analysis, Embedded Software.

1 Introduction

A watermark is an identifying information that is embedded in media. Watermarking is of dual use: first, it discourages intellectual property (IP) theft; and second, when such theft has occurred, it allows to prove ownership [13]. The attacker aims to garble the watermark by applying simple transformations of the watermarked media such as additions, distortions, and deletions. The verifier performs a specific test to prove the existence of the watermark, even if it is degraded due to simple transformations.

This paper addresses IP protection of embedded software for which a simple read-out of a suspected program code from the memory of a microcontroller is prevented. The main focus is on authorship watermarks that embed information identifying the author of the software. We act on the software watermark model that has been introduced by Becker et al. [10]. In this model, the verifier needs to have physical access to the device that runs the suspected software. The verifier does not have direct access to the program code of the suspected software. In contrast to previous work, we use the fault-channel of a device to construct a new watermarking scheme. To our best knowledge, this work is the first to implement fault-channels watermarks.

2 Related work

2.1 Fault Analysis

Accidental computation errors in chips caused by high-energetic particles are well known since the 1970s [6]. Since the pioneering work of Boneh et al. [12] a new area of research dedicated to fault attacks on cryptographic implementations has emerged. These kind of attacks aim at determining secret or private cryptographic keys based on erroneous outputs of the implementations. Originally, these attacks were theoretical. Anderson and Kuhn [1,2] reported that fault injections using voltage drops and glitch attacks are indeed practical and already in use in pay-TV hacks to manipulate the program counter of a CPU. A new breakthrough in fault injection was demonstrated by Skorobogatov who uses the impact of light on a de-packaged chip [21]. Until now, the use of laser based fault injection is state-of-the-art. A valuable survey of fault injection techniques and countermeasures is provided in [6].

Microcontrollers that were investigated on non-invasive fault injections have turned out to be vulnerable. Closely related to our work is [5], in which the fault effects of clock glitches of the ATMega163 microcontroller were tested in depth. Similarly, specific instructions of the microcontroller platforms Atmel ATxmega256 and ARM Cortex-M0 were investigated in [17]. Other works characterize the effects of low voltage attacks on the ARM9 microprocessor [9,8].

Secure smartcard chips are usually equipped with effective countermeasures on non-invasive fault attacks such as voltage and clock sensors and usage of an internal clock. Defenses against fault injection using laser light, however, are much more difficult. Because of this we act on the assumption that there are some vulnerabilities left even on a security chip that can be used for semi-invasive fault injection. Our approach on watermarking requires that faults can be introduced, no matter how they are introduced. Even if cryptographic implementations include fault detection mechanisms in software, the execution of such routines can be of use for our watermarking scheme.

2.2 Side-Channel Watermarks

Becker et al. [11,10] propose to use side-channel leakage for watermark detection. It is built upon Differential Power Analysis (DPA) [16]. The watermark is realized by adding extra code for a watermark key, a combination function, and a leakage generator. The leakage generator transmits the result of the combination function through a side channel leakage and it is detected in the same way as with DPA. A main drawback of these schemes is the need for additional code or hardware which is at risk to be localized by reverse engineering of the adversary. Obfuscation can help increasing the efforts for reverse engineering but perfect obfuscation is impossible [7]. A further drawback is that these side-channel watermarks need to be always-on, thereby consuming additional power.

Another side channel approach to watermarking by Durvaux et al. [14] uses soft physical hash functions. This approach is quite similar to ours, as it uses the software implementations as they are, without additional code due to watermarks. The

authors use the correlation coefficient as similarity score. For this, they need to transform the side channel traces of the two implementations under test to a fixed length vector. For compression, removal of noisy parts and subsequent fast Fourier transform (FFT) of the data is proposed. From the frequency domain data, the authors extract the values corresponding to frequencies below the clock frequency from both vectors and use them for the similarity score. Experimental results in [14] show that the robustness of this approach is rather low as it is susceptible to the addition of dummy instructions.

Research in building a side-channel based disassembler goes back to the initiating work of Quisquater and Samyde [20]. It aims at extracting executed instructions from the side-channel leakage. Recent work by Strobel et al. [23] reported a 87.69% success rate on a real application on a PIC16F687 microcontroller by training a classifier with electromagnetic traces obtained by applying multiple EM-probes at different positions of the chip simultaneously.

3 Fault-Channel Watermarks

A fault-channel watermark consists of a well-characterized sequence of fault-sensitive instructions that are embedded in the functional code. It is an important difference to previous work that such a watermark design is part of the functional code, i.e. additional code for the watermark is not needed anymore.

Security Objective Robustness is the main security objective for authorship watermarks. A watermarking algorithm is robust if it is legible after all possible transformations [18].

Adversary Model We assume that the adversary obtains the program code of the original embedded software under IP protection. The adversary can insert, delete, and substitute assembly instructions. The adversary may embed the original program code inside a software wrapper, thereby hiding the external interfaces of the original program.

Same as in other state of the art approaches, e.g., [10], our adversary model does not include compiler-based code transformations that require a decompiler for transforming the program code to a high-level programming language.

Verifier Model We assume that a read-out prevention makes it impossible to directly extract the binary of the code from the memory of the microcontroller. The verifier is assumed to have physical access to the microcontroller and is assumed to be able to repeatedly trigger execution of the suspected code. IP protection and verification is done in three steps: preparation, embedding, and verification.

3.1 Preparation: Characterization of Fault Sensitivity

During the preparation step, a microcontroller architecture is analyzed for its susceptibility to fault attacks. Such an analysis requires equipment for fault injection.

As we aim at a cheap and easy-to-setup non-invasive fault injection, methods such as voltage dropping and glitching are preferred. However, this does not exclude more advanced characterization, e.g., using laser light. The aim is to detect and parameterize easily reproducible faults of the microcontroller. The preparation step may include a profiling of specific assembly instructions. Alternatively, existing program code can be scanned with different fault injection parameters at execution time. Using a simulator that maps the fault injection time to the executed instruction, the fault sensitive instruction in the code can be identified.

As result of the preparation step, the designer knows about appropriate fault parameterizations and some fault sensitive assembly instructions, which are a subset of the microcontroller instruction set. For each instruction, an appropriate parameter set for fault induction may be stored.

Example 1. In case of voltage-controllable faults, the parameter set includes fault probability, fault voltage, timing offset within an instruction cycle and timing pulse width.

Example 2. In case of clock-controllable faults, the parameter set includes fault probability and glitch period.

Example 3. In case of laser-controllable faults, the parameter set includes fault probability, XY-coordinates of the chip, timing offset within an instruction cycle, laser wavelength, laser pulse width, and laser energy.

3.2 Embedding

Embedding of the fault sensitive instructions and thereby the watermark occurs during software development. A reference watermark of the finalized implementation is based on a chosen fault injection parameter set of the preparation phase. The implementation is entirely scanned as a function of the timing offset where the fault injection applies. More precisely, the implementation is invoked for each fault injection, and the output of the implementation is recorded. For the next fault injection, the timing offset is increased by a fixed amount that is at maximum one clock period, so that each instruction is tested with at least one fault injection.

As result of each scan, a string is output that denotes the watermark. Each string consists of characters from the same alphabet. For simplicity, we assume the set of {0,1,2}, where '0' indicates no error, '1' indicates a data output error and '2' indicates an unexpected end of computation (program crash). Table 1 shows an example of a fault injection scan and the resulting string.

Multiple scans of the implementation can be used to build a fault-channel profile of the implementation. For each time offset, the profile specifies the frequency that each character appears [15], i.e., the profile is a two dimensional matrix $P_{y,j}$ for each character y of the alphabet and each timing offset j.

3.3 Verification

The objective of the verifier is to decide whether a microcontroller under test contains relevant parts of the reference watermark from the embedding phase.

Table 1. An excerpt from a fault injection scan. Fault injection is applied once every clock cycle. An `LDI` instruction takes 1 clock cycle to execute, a `CALL` instruction needs 4 clock cycles, and a `PUSH` needs 2 clock cycles. The scan produces the string `'112000002'`.

Offset (ns)	Instruction	Response to fault injection	Fault injection string
400	`LDI`	data output error	1
900	`LDI`	data output error	1
1400	`LDI`	program crash	2
1900	`CALL`	no error	0
2400		no error	0
2900		no error	0
3400		no error	0
3900	`PUSH`	no error	0
4400		program crash	2

The verifier entirely scans the executable parts of the suspected code with the fault injection parameter set of the embedding phase. For each entire scan, the verifier outputs a string that is built in the same way as in the embedding phase.

Finally, the verifier compares the similarity between the obtained string and the original watermark profile, cf. Section 3.3. The existence of the original watermark is successfully proven, if the similarity of the watermark is sufficiently high.

Edit Distance In order to assess the reproducibility of our results using the same implementation as well as to assess the success of watermark detection when comparing the results with possibly different implementations obtained in the embedding and verification phase, we need to define how we quantitatively compare strings. During the preparation tests, the strings are generally of the same length, however, this does not always hold. In the more general case, we want to compare the string S_1 with length m to the string S_2 with length n. For convenience, we assume that a string starts with the index 1 (as opposed to 0).

The edit distance d_e between two strings is defined as the minimum number of edit operations—insertions, deletions, and substitutions—needed to transform the first string into the second [15] or vice versa. Two strings are said to be similar if their edit distance is significantly small. This concept goes back to Levenshtein and the Levenshtein distance is generally recognized as a famous representative of edit functions, though it is not consistently used in the literature [22].

Algorithm 1 shows how to compute normalized edit distance. The function $t(i, j)$ computes whether the characters $S_1(i)$ and $S_2(j)$ match and assigns the cost 0 when they do, otherwise the character $S_1(i)$ should be substituted by character $S_2(j)$, in which case the cost is 1. Matrix D saves the traceback that can be used to recover the sequence of steps for optimally transforming S_1 into S_2. The computed edit distance is normalized by the maximum length of both strings, so that the resulting numbers are in range between 0 and 1 and can be compared to each other.

Algorithm 1: Computation of edit distance of transforming S_1 into S_2

Input: Strings S_1 and S_2 with lengths m and n, respectively
Output: d_e - normalized edit distance between S_1 and S_2

```
1  D =zeros(m+1, n+1)
2  for i = 1 : m do
3      D[i, 0] = i
4  for j = 1 : n do
5      D[0, j] = j
6  for i = 1 : m do
7      for j = 1 : n do
8          D[i, j] = min(D[i−1, j−1] + t(i, j),      // match or substitute S1(i) and S2(j)
9                        D[i−1, j    ] + 1,          // delete S1(i) with the cost of 1
10                       D[i    , j−1] + 1)          // insert S2(i) with the cost of 1
11 return d_e = D[m,n] / max(m,n)
```

The time and space complexities of Algorithm 1 are in $O(mn)$. However, in our case, it is not necessary to know *how* to transform one string into another, but only how much it costs. Thus, the space requirement can be reduced to $O(m)$, as two rows of matrix D are sufficient to compute the edit distance.

4 Experimental Setup

Figure 1 shows the high-level view of our setup.

ATMega163 Smartcard In this work we use a smartcard with an ATMega163 microcontroller [3]. It is an 8-bit RISC microcontroller that has 32 general-purpose registers, 16K bytes flash, and 1024 bytes internal SRAM. It can be driven by a clock with frequencies between 0 and 8 MHz. It features 130 instructions, most of which run in 1 clock cycle [4]. The processor uses two-stage pipelining, so that during execution of one instruction, the next instruction is fetched and decoded.

Before an application can be run on the smartcard, it has to be compiled. This is accomplished using avr-gcc, which produces a hex-file that can be loaded into the flash of the smartcard using a smartcard programmer. Our applications for the smartcard are written in combination of C and assembly language.

GIAnT The generic implementation analysis toolkit (GIAnT) is an FPGA-based board for fault injection and side channel analysis [19]. Figure 2 shows the board. The GIAnT board was built using a modular approach, so that each component can be bought separately. The core of the board is a Spartan 6 FPGA that is responsible for communication with the device under test, and performing the fault injections. The board has a slot for inserting a smartcard and can communicate with it using T=0 and T=1 protocols. The board can control the power supply of the smartcard and

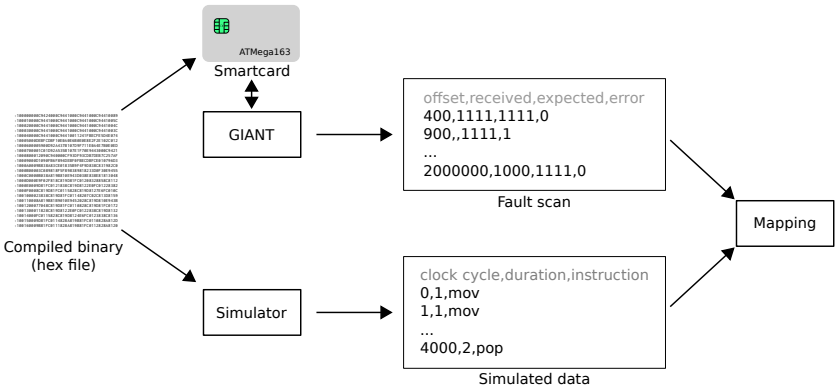

Fig. 1. Data flow of a fault injection scan.

induce a voltage drop upon request. The GIAnT board was configured to run the smartcards at 2MHz.

The user can set several parameters, such as the *trigger* type, time *offset* from the trigger after which the fault injection should occur, the *pulse width* of the injection, the *voltage level* during the fault injection, and even a complex voltage pattern with different voltage levels. There are several trigger types: a trigger upon the first input to the smartcard, a trigger upon the first output from the smartcard, and a trigger upon a "HIGH" voltage applied to the programming pin of the smartcard.

To perform a fault injection, the user has to program the GIAnT board, reset the smartcard, feed it with all necessary data to initialize the start up sequence, as the smartcard is reset before doing a fault injection. When the board is ready, the trigger needs to be set, after which the application under test can be executed. The GIAnT board reacts to the trigger and makes the voltage of the smartcard drop at the specified time offset after the trigger for the specified pulse width.

Fault Injection Scan In this work, we want to perform fault injection scans of suspicious applications. During one such scan, each instruction is disturbed at least once, and the offset is increased by a fixed step size. The result of a fault scan is a dataset that contains the offset of each fault injection performed, the received output, the expected output, and whether or not the fault injection cause an error such that the smartcard failed to respond.

ATMega163 Simulator To understand the impact of a fault injection on any arbitrary instruction, we need to know at what time each instruction is executed. We achieved this by writing an ATMega163 simulator that can load and run an arbitrary hex file compiled for an ATMega163 microcontroller. Our simulator logs executed instructions, the internal state of the CPU, the current number of clock cycles, and the number of clock cycles that an instruction requires to execute.

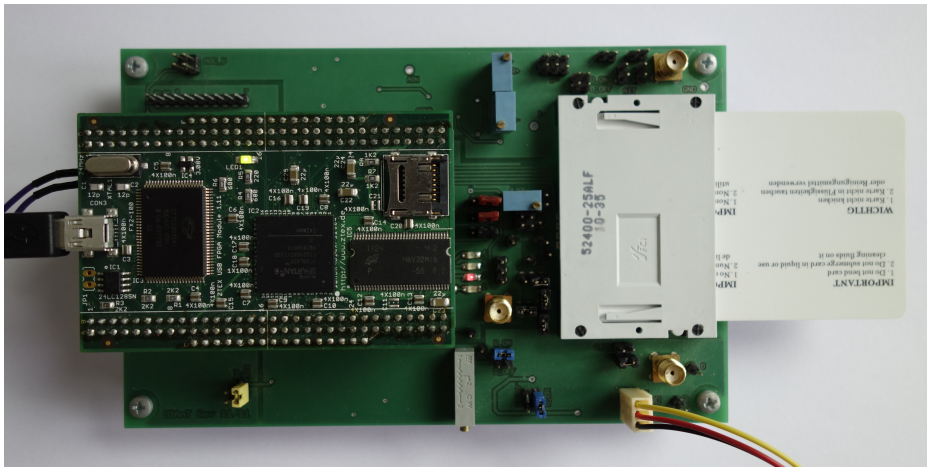

Fig. 2. The GIAnT board.

Mapping Injections to Instructions The final step is to combine the data obtained from running the fault injection scan with the data from the simulator. The mapping gives us the insight of which instructions are vulnerable to fault injection and at what offset. Further, it allows us to see the dependencies between the opcode, the internal state of the processor, and the applied fault injection.

5 Experimental Results

For our experimental analysis we used several different AES-128 program codes on the ATMega163. Two AES program codes (AES0 and AES1 versions) stem from source codes written in assembly and three program codes (AES2 versions) were generated from one AES source code in C using different compiler versions and options. All AES implementations are embedded into a minimized smartcard operating system that allows to set the AES key and AES plaintext. The AES key and plaintext are fixed for the duration of whole fault scan. One fault injection scan over all 12010 clock cycles of AES2-v-0 implementation requires around 3.3 hours, and to scan all 4480 clock cycles of AES1-v-0 requires around 1 hour.

Table 2 shows an overview of all implementations and the parameters used to perform fault injection scans. There are three versions of the AES1 implementation: AES1-v-0 and AES1-v-1 use the same program code but different key and plaintext and AES1-v-2 is a modified program code version of AES1 that is generated by adding dummy assembly instructions throughout the whole code.

5.1 Success Probability of Fault Injection

Figure 3 illustrates average success probabilities obtained by applying a fault injection scan of AES0 implementation for each individual assembly instruction at

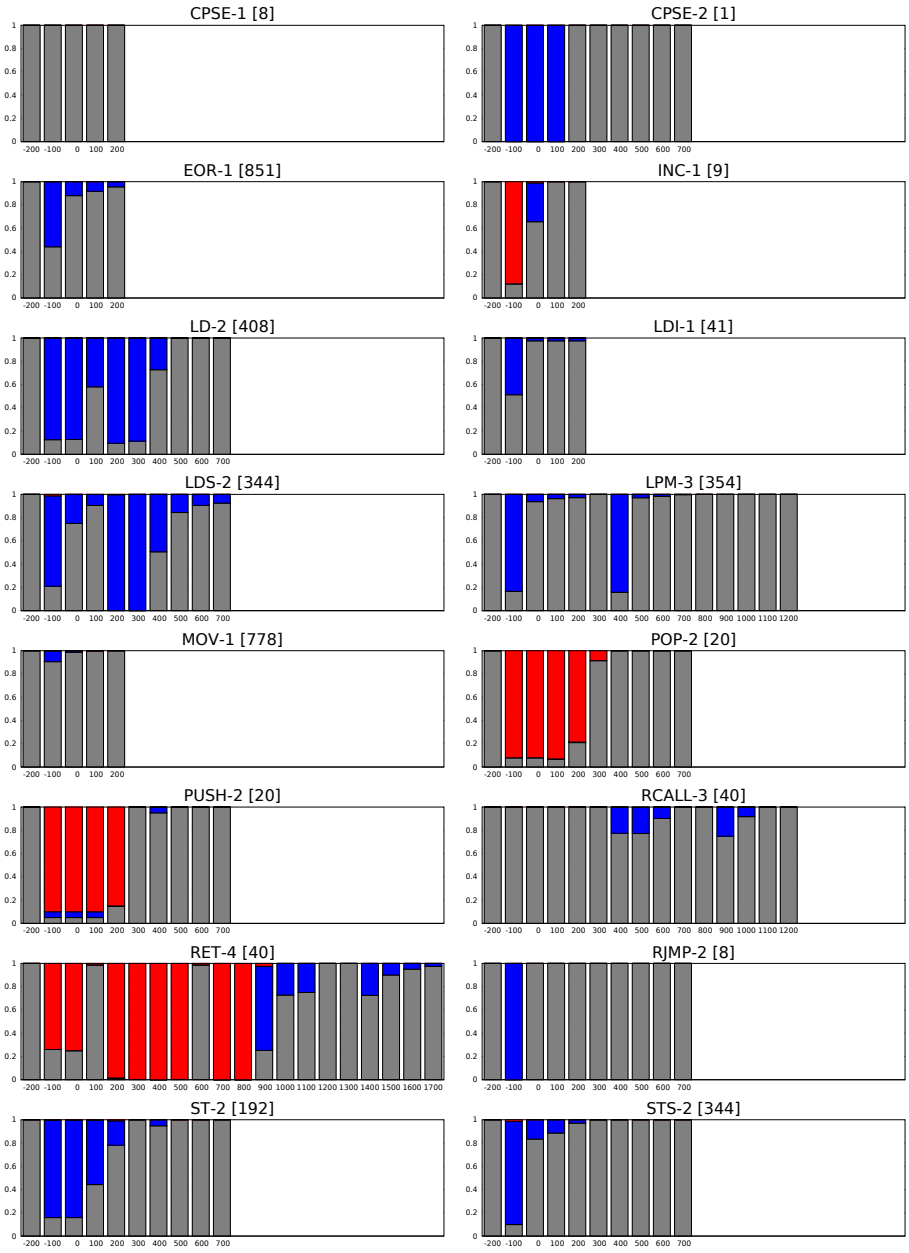

Fig. 3. Probability of fault injection success for all 15 instructions of the AES0 implementation at different offsets. The name of each instruction is followed by the number of clock cycles it takes to execute it. The number of times the instruction is executed during one AES encryption is shown in square brackets. Horizontal axis denotes the offset in nanoseconds, and vertical axis the probability. Fault injections that were not successful are shown in white, fault injections that caused data output errors are shown in gray, and fault injections that resulted in program crash are shown in black.

Table 2. Scanned AES implementations, their versions, and measurement details.

Name	AES0	AES1-v-0	AES1-v-1	AES1-v-2	AES2-v-0	AES2-v-1	AES2-v-2
Language	assembly	assembly	assembly	assembly	C	C	C
Optimization	-	-	-	-	-O3	-O3	-O2
Compiler version	-	-	-	-	4.8.4	4.3.3	4.3.3
N. of clock cycles	5705	4480	4480	5569	12010	12006	21980
N. of instructions	15	28	28	32	38	32	38
Inj. step size	100 ns	100 ns	500 ns	500 ns	500 ns	500 ns	500 ns
Inj. pulse width	500 ns	500 ns	500 ns	500 ns	500 ns	500 ns	500 ns
N. of scans	10	10	5	5	10	10	10
All key bytes	0x0a	0x0a	random	0x0a	0x0a	0x0a	0x0a
All plaintext bytes	0x09	0x09	random	0x09	0x09	0x09	0x09

different offsets. The step size of the fault injection offset is 100 ns, which means that there are 5 fault injections per clock cycle. Note that the `CPSE` instruction needs one clock cycle if the two compared registers are not equal, and two clock cycles if they are equal, in which case the next instruction will be skipped. It was used to check whether the final AES round was reached, so that its two-clock-cycles version is seen only once.

The statistics were computed by aligning all executed instructions and computing the average success at each injection offset. We assume that if an injection occurs during the last 200 ns of an instruction, it contributes to the injection statistics of the next instruction. This is because the pulse width of the injection is 500 ns, which is the same as the clock period the ATMega163. It means that if the injection starts 200 ns before an instruction, it disturbs the previous instruction for 200 ns, and the current one for 300 ns.

From Figure 3, we learn that a fault injection is most successful when applied 100 ns before each clock cycle. We used this insight to reduce the duration of fault injection scans by setting the offset step size to 500 ns and starting the scan at 400 ns.

5.2 Repeatability

First, we evaluate the amount of noise that occurs while repeating the same AES implementation using the same key and input data. For AES0 and AES1-v-0 we carried out $r = 10$ repetitions of fault injection scans using constant input and key data. The fault injections were applied five times per clock cycle with an offset increment of 100 ns. Table 3 summarizes the probabilities of error-free execution (state 0), data output error (state 1) and program crash (state 2). It can be seen that the amount of observed errors sums up to about 20% and 25%, respectively.

We computed the edit distances of all $\frac{r(r-1)}{2}$ pairwise combinations of single strings S_i and S_j with $i \neq j$, for each implementation. For AES0, the average edit distance is equal to $\overline{d_e}(S_i, S_j) \approx 62.8 \pm 6.1$. For AES1-v-0, it is $\overline{d_e}(S_i, S_j) \approx 41.6 \pm 5.3$.

Table 3. Fault probabilities of the AES implementations.

Impl.	Probability state '0'	Probability state '1'	Probability state '2'
AES0	0.7892	0.1941	0.0167
AES1-v-0	0.7342	0.2464	0.0194

Additionally, we computed *majority strings* of the two implementations. We define the *majority string* $\overline{S}$ as follows: for each time offset, it contains the most probable state. In case that several states occur equally frequently, we decide in favor of the more probable state of the entire implementation. Doing so, we further can reduce the noise comparing single scans and the majority string of 9 scans, cf. Table 4.

Table 4. Fault characteristics of the AES implementations.

Impl.	No. fault injections	$\overline{d_e}(S_i, S_j)$	$\overline{d_e}(S_i, \overline{S})$
AES0	28550	62.8 ± 6.1	38.0 ± 6.4
AES1-v-0	22500	41.6 ± 5.3	26.7 ± 4.5

We conclude that using constant data, the noise is low. For a string length of 28550 for AES0, we can observe 62.8 mismatches on average when comparing single scans, which is a probability of approximately 0.002. AES1-v-0 has an even lower mismatch probability.

5.3 Comparing Different AES Implementations

Table 5 shows a comparison of all AES implementations. Originally, AES0 and AES1-v-0 were scanned with offset step size of 100 ns, while the other implementations with 500 ns. To compare them to each other, the number of data points of AES0 and AES1-v-0 was reduced to only include injection offsets starting from 400 ns with injection step size of 500 ns.

Self-distance Normalized edit distance of each fault injection scan of an implementation to all other scans of the same implementation is small, and significantly smaller than to the scans of all the other implementations. This means that the edit distance is especially well-suited for revealing binaries that were used without modification.

Different input data Implementations AES1-v-0 and AES1-v-1 are identical, however, they are invoked with different key and plaintext. While AES1-v-0 has all key bytes set to `0x0a`, and plaintext bytes to `0x09`, the AES1-v-1 implementation has the key equal to `0x0b56e9b99f17ee9bc7cacbfbc6e7b1f2`, and plaintext set to

Table 5. Mean normalized edit distances computed pairwise for all measurements.

	AES0	AES1-v-0	AES1-v-1	AES1-v-2	AES2-v-0	AES2-v-1	AES2-v-2
AES0	**0.0032**	0.3537	0.3502	0.3506	0.5281	0.5342	0.7404
AES1-v-0	0.3537	**0.0015**	0.1116	0.2623	0.6272	0.6307	0.7954
AES1-v-1	0.3502	0.1116	**0.0441**	0.2972	0.6269	0.6309	0.7954
AES1-v-2	0.3506	0.2623	0.2972	**0.0288**	0.5529	0.5617	0.7454
AES2-v-0	0.5281	0.6272	0.6269	0.5529	**0.0131**	0.3389	0.4815
AES2-v-1	0.5342	0.6307	0.6309	0.5617	0.3389	**0.0462**	0.4738
AES2-v-2	0.7404	0.7954	0.7954	0.7454	0.4815	0.4738	**0.0169**

`0x1d4c2c6300b72ee1b094717c29f46c7d`. The normalized edit distance between these two implementations is higher than their respective self-distances, but closer each other than to all other implementations. To minimize the impact of processed data, the IP verifier should use the same data during verification. In situations where it is not possible (for example, the secret key can be set to a fixed value by the attacker), the verifier can precompute a majority string of the original implementation by feeding the algorithm with random data at each fault injection.

Robustness to dummy instructions The AES1-v-2 implementation was designed to simulate an attacker that has taken the original AES1-v-0 implementation and added a significant number of dummy instructions. These instructions are combined in a way that they have no impact on the state, unless the application is scanned by fault injections. For example, it is possible to push an important register onto the stack, modify it, and pop it back, as shown in Listing 1.1. Under normal circumstances, these operations will not have any impact on subsequent computations. However, a successful fault injection on `PUSH` or `POP` instructions will change the data in some registers. Since there is always only one injection per program execution, a fault can cause either `PUSH` or `POP` to malfunction, so that data used in subsequent computations will be different from the data of the original implementation. Thus, dummy instructions will only add extra characters to the fault injection string at their respective offsets, but will not cause changes in the original fault injection string at other offsets.

In addition to `PUSH` and `POP` instructions, several other *inverse-instructions-pairs* have been used, such as, e.g., `NEG-NEG`, `EOR-EOR`, `INC-DEC`. These instructions, when applied to one and the same register in succession, cause no changes under normal circumstances. However, a successful fault injection will change the content of the register, and influence the remaining computations.

In total, 1029 extra clock cycles have been added in various places throughout the code: in the key scheduler, in MixColumns, in SubBytes, in and outside of the AES rounds. This makes 22.97% of added dummy clock cycles over the 4480 clock cycles of the initial implementation.

We observed the average normalized edit distance of 0.26 between the original and the modified implementation. Despite all additional instructions, the normalized edit distance between the two implementations is smaller than their normalized edit distances to all other implementations. These results show that our approach is still robust given an amount of 22.97% added dummy clock cycles.

```
NOP
PUSH r0
PUSH r18
PUSH r19
PUSH r30
LDI  r18, 0xff ;; r18 := 0xff
MOV  r0,  r18  ;; r0  := r18
LDI  r19, 0x7c ;; r19 := 0x7c
EOR  r30, r0   ;; r30 := r30 xor r0
EOR  r19, r18  ;; r19 := r19 xor r18
MOV  30,  r19  ;; r0  := r18
POP  r30
POP  r19
POP  r18
POP  r0
```

Listing 1.1. Dummy instructions in the key scheduler.

Durvaux et al. [14] added 52 dummy clock cycles to the total of 2739 clock cycles of the *Furious* implementation of AES, which makes 1.9% of extra clock cycles. In response, the similarity score changed from 0.9 to 0.55. This might be caused by the use of Pearson's correlation to compute the similarity score between power traces—the correlation is not robust when additional instructions are inserted at several different positions in the code. Furthermore, correlation compares two vectors that must have the same size. Our approach, however, uses edit distance that allows comparison of arbitrary-sized strings. In addition, it is possible to extract matching substrings and see which parts of the original code have been used in the modified implementation.

Compiler options and versions AES2-v-0 and AES2-v-1 stem from one and the same implementation written in C that was compiled using different versions of the avr-gcc compiler. Their duration deviates by only 4 clock cycles, but the generated assembly code is significantly different. This is also reflected in their normalized edit distance—it is much larger then their respective self-distances. However, their mutual normalized edit distance is smaller than their normalized distance to all the other implementations.

The AES2-v-2 implementation was compiled using a different optimization option: -O2, as opposed to -O3. This resulted in larger run time of the AES encryption. This implementation has a low self-distance and a high distance to all other implementations.

6 Conclusion and Future Work

In this work we have presented a new method for software IP protection based on fault analysis that does not require software developers to change or modify their implementations. This method was designed to enable analysis of IP that resides in read-protected memory. To verify whether a suspicious system uses software legitimately, we perform a fault injection scan of the entire implementation, convert the results of the scan into a string, and compare the resulting string to a string obtained from scanning our own implementation. The two strings are compared using normalized edit distance, which tells us how many changes we must apply to one string to transform it into the other.

The method was evaluated by comparing several AES implementations written for an ATMega163 microcontroller. Experimental results show that the method is especially well-suited for comparing compiled binaries and implementations in assembly code that were taken without modifications. There is little variation between the fault injection scans of the same implementation. Different implementations have a high normalized edit distance from each other, such that it is very unlikely that one implementation gets mistaken for another.

The strength of using normalized edit distance becomes apparent when we randomly add dummy instructions to the original code that do not have impact on the state unless a fault injection disturbs their operation. Adding dummy instructions results in an increase of normalized edit distance that is proportional to the number of instructions added normalized by the overall size of the original code. Thus, for 22.97% additional clock cycles, the normalized edit distance increased of the original and modified AES implementation was equal to 0.26, which means that 26% of one implementation have to be changed in order to transform it into the other. The robustness of our method is significantly better than the method of [14] that is based on Pearson's correlation, whose performance decreases substantially after adding a very small number of dummy clock cycles.

Edit distance is a global alignment method that compares two strings as a whole and might ignore high local similarity in favor of a better global alignment [15]. A promising direction for future work is the evaluation of local alignment methods that can find substrings of high similarity, which would allow us to identify sub-parts copied from the original code.

Acknowledgments. This work has been supported in parts by the German Federal Ministry of Education and Research (BMBF) through the project DePlagEmSoft, FKZ 03FH015I3.

References

1. ANDERSON, R. J., AND KUHN, M. G. Tamper Resistance – a Cautionary Note. In *The Second USENIX Workshop on Electronic Commerce Proceedings* (1996), pp. 1–11.
2. ANDERSON, R. J., AND KUHN, M. G. Low Cost Attacks on Tamper Resistant Devices. In *Security Protocols, 5th International Workshop* (1997), pp. 125–136.

3. ATMEL. *ATmega163(L) Datasheet (revision E)*, Feb. 2003.
4. ATMEL. *Atmel AVR 8-bit Instruction Set Manual (revision 0856J)*, July 2014.
5. BALASCH, J., GIERLICHS, B., AND VERBAUWHEDE, I. An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. In *FDTC 2011* (2011), pp. 105–114.
6. BAR-EL, H., CHOUKRI, H., NACCACHE, D., TUNSTALL, M., AND WHELAN, C. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE 94*, 2 (Feb 2006), 370–382.
7. BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S. P., AND YANG, K. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology - CRYPTO 2001* (2001), pp. 1–18.
8. BARENGHI, A., BERTONI, G., PARRINELLO, E., AND PELOSI, G. Low Voltage Fault Attacks on the RSA Cryptosystem. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on* (Sept 2009), pp. 23–31.
9. BARENGHI, A., BERTONI, G. M., BREVEGLIERI, L., PELLICIOLI, M., AND PELOSI, G. Low Voltage Fault Attacks to AES. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on* (June 2010), pp. 7–12.
10. BECKER, G. T., BURLESON, W., AND PAAR, C. Side-Channel Watermarks for Embedded Software. *9th IEEE NEWCAS Conference* (2011).
11. BECKER, G. T., STROBEL, D., PAAR, C., AND BURLESON, W. Detecting Software Theft in Embedded Systems: A Side-Channel Approach. *IEEE Transactions on Information Forensics and Security 7*, 4 (2012), 1144–1154.
12. BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *Advances in Cryptology - EUROCRYPT '97* (1997), pp. 37–51.
13. COLLBERG, C. S., AND THOMBORSON, C. D. Software Watermarking: Models and Dynamic Embeddings. In *POPL* (1999), pp. 311–324.
14. DURVAUX, F., GÉRARD, B., KERCKHOF, S., KOEUNE, F., AND STANDAERT, F. Intellectual Property Protection for Integrated Systems Using Soft Physical Hash Functions. In *Information Security Applications - 13th International Workshop, WISA 2012* (2012), pp. 208–225.
15. GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1999.
16. KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *CRYPTO* (1999), pp. 388–397.
17. KORAK, T., AND HOEFLER, M. On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014* (2014), pp. 8–17.
18. NAGRA, J., THOMBORSON, C. D., AND COLLBERG, C. S. A Functional Taxonomy for Software Watermarking. In *ACSC* (2002), pp. 177–186.
19. OSWALD, D. GIAnT: Generic Implementation ANalysis Toolkit. SourceForge, 2014.
20. QUISQUATER, J., AND SAMYDE, D. Automatic Code Recognition for Smartcards Using a Kohonen Neural Network. In *Proceedings of the Fifth Smart Card Research and Advanced Application Conference, CARDIS '02* (2002).
21. SKOROBOGATOV, S. P., AND ANDERSON, R. J. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002* (2002), pp. 2–12.
22. SMYTH, B. *Computing Patterns in Strings*. Pearson Education Limited, 2003.
23. STROBEL, D., BACHE, F., OSWALD, D., SCHELLENBERG, F., AND PAAR, C. SCANDALee: A Side-ChANnel-based DisAssembLer using Local Electromagnetic Emanations. In *Design, Automation, and Test in Europe (DATE)* (March 9–13 2015).