Detecting Similar Code Segments through Side Channel Leakage in Microcontrollers

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

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

Abstract. We present new methods for detecting plagiarized code segments using side-channel leakage of microcontrollers. Our approach uses the dependency of side-channel leakage on processed data and requires that the implementation under test accepts varying known input data. Detection tools are built upon a similarity matrix that contains the absolute correlation coefficient for each combination of time samples of the two possibly different implementations as result of side channel measurements. These methods are evaluated on smartcards with ATMega163 microcontroller using different test applications written in assembly language. We show that our methods are highly robust even against a skilled adversary who modifies the original assembly code in various ways. Our approach is non-intrusive, so that the application does not need to be additionally watermarked in order to be protected—the resulting pattern of data leakage of the microcontroller executing the code is considered as its own watermark.

Keywords: Side-Channel Watermarking, IP Protection, Code Similarity Analysis, Similarity Matrix, Software Reverse Engineering, Embedded Software.

1 Introduction

Intellectual property (IP) theft of embedded software for microcontrollers with effective read-out protection of memories constitutes a hard problem for IP protection nowadays. A direct binary analysis is not feasible as the suspicious program code needs to be physically extracted from the internal code memory first which constitutes a hard and cost-intensive problem in practice. To address this problem, the use of side channel leakage was first proposed by Becker et al. [1] by implementing additional watermarking code for a leakage generator. In their followup work, Becker et al. [2] have used the coincidence that ATMega8 microcontrollers leak the Hamming weight of the opcode when an instruction is fetched. The power traces are converted into strings, and the strings are compared using string matching algorithms, such as edit distance and Boyer-Moore-Horspool algorithm. However, this approach relies on processor-specific properties that do not in general apply—not all microcontrollers leak Hamming weights of the opcodes. In addition, Hamming weights of opcodes are not unique, such that many different instructions can have the same Hamming weight. Also, the method is not robust against code-transformation attacks that exchange assembly instructions by others leading to the same output, replace registers and RAM and flash addresses of variables and data.

A different approach was taken by Strobel et al. [3], where the authors tried to disassemble instructions from electromagnetic (EM) traces of PIC16F687 microcontroller by

training a classifier that is able to distinguish 87.60% of instruction classes correctly. This approach, however, has been only tested on a small range of microcontrollers.

Another way to prove ownership is to look for dependency between the power consumption and executed code. Durvaux et al. [4] compute similarity of two power traces using Pearson's correlation along the time axis. The intrinsic side channel leakage of different implementations is compared without the need of additional watermarking code. However, their proposed method shows low robustness if the adversary adds dummy instructions to the IP protected code.

In this work we propose a new method that addresses the software plagiarism problem on a much finer level—our method can not only identify whether the whole program is a plagiarism, but also which code segments have been plagiarized. In addition, our method shows a significantly better robustness against additions of dummy code when compared to the current state of the art approach.

2 Our Approach

Since the pioneering work of Kocher et al. [5] it is well known that during code execution data dependencies in microcontroller programs induce side-channel leakage that is measurable in the power consumption of the microcontroller or in its electromagnetic (EM) emanation. And side-channel leakage discloses the positions in time where a targeted intermediate data item is processed, such that repeatedly processing the same data using the same implementation will result in a very similar physical leakage. Our approach builds upon this finding for the use in IP protection. Considering the framework for IP protection of [6], our contribution introduces new detection tools for IP protection using side channel leakage. Our main approach requires that an equivalent input data channel is available for the configuration of the program code of the genuine implementation and the second unknown and possibly suspicious implementation. The existence of an equivalent input data channel is a reasonable assumption when different program code is used for the same purpose. Our objective is to provide robust similarity detection tools using code and data characteristics in two given implementations.

2.1 Extraction: Data Acquisition

The objective of the data acquisition step is the collection of N side channel traces of the genuine program and N side channel traces of the unknown program, both processing the same list of N varying input data. This can be achieved either in a chosen-input scenario but also in a known-input scenario. In the latter case the unknown program has to be measured first and a chosen-input variant of the genuine program has to be available that is fed with the same input data afterwards. Each set of measurements and their corresponding input data represent a soft physical hash value [6] or a fingerprint of the genuine and unknown code, respectively. To make comparison of similarity easier, it is recommended to use an identical measurement setup for both implementations. Details on side-channel measurement setups can be found in [5, 7].

2.2 Extraction: Preprocessing

Before comparing the two fingerprints, the number of samples per trace is reduced by compressing them, e.g., by extracting the mean of each clock cycle. In principle, this preprocess-

ing step is optional, but useful in practice to reduce the computation time in the following detection phase.

2.3 Detection: Similarity Matrix

As result of the extraction phase, we build two matrices:

1. An $N \times M_1$ matrix $T_{genuine}$ that contains N rows of compressed side channel traces with M_1 samples each. $T_{genuine}$ contains the extracted measurement data from the genuine implementation.
2. An $N \times M_2$ matrix $T_{unknown}$ that contains N rows of compressed side channel traces with M_2 samples each. $T_{unknown}$ contains the extracted measurement data from the unknown implementation.

In the following, we denote an entry of a matrix T as T_{ij}, the j-th column vector as $T_{:,j}$, and the i-th row vector as $T_{i,:}$. An entry of a vector $\boldsymbol{x}$ is denoted by x_i and $\boldsymbol{x}^T$ is the transpose of $\boldsymbol{x}$.

For the computation of similarity between the data leakage of the genuine and the unknown code the sample Pearson correlation coefficient

$$\hat{\rho}(\boldsymbol{x},\boldsymbol{y}) = \frac{\Sigma_{i=1}^{N}(x_i - \hat{x})(y_i - \hat{y})}{\hat{\sigma}_x \hat{\sigma}_y}$$

is used, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of length N, $\hat{x}$ and $\hat{y}$ are the sample means of $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, and $\hat{\sigma}_x, \hat{\sigma}_y$ are their respective sample standard deviations.

The basis for our detection methods is the $M_1 \times M_2$ similarity matrix S. Each entry S_{ij} is the absolute correlation coefficient of the column vector $T_{genuine_{:,i}}$ and the column vector $T_{unknown_{:,j}}$.

$$S = \begin{pmatrix} |\hat{\rho}(T_{genuine_{:,1}}, T_{unknown_{:,1}})| & \cdots & |\hat{\rho}(T_{genuine_{:,1}}, T_{unknown_{:,M_2}})| \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ |\hat{\rho}(T_{genuine_{:,M_1}}, T_{unknown_{:,1}})| & \cdots & |\hat{\rho}(T_{genuine_{:,M_1}}, T_{unknown_{:,M_2}})| \end{pmatrix}$$

It holds $0 \leq S_{ij} \leq 1$ for all entries of S. If S_{ij} is close to 1 this indicates a high similarity between sample point i in the genuine implementation and sample point j in the unknown implementation, whereas entries with S_{ij} close to 0 indicate that there is no similarity at these two sample points.

In the final step, we need to make a decision whether the two implementations are similar or not as a whole, and whether they contain highly similar subparts.

Detection Tool: Visual Inspection The similarity matrix pinpoints the sequence of data processing in the unknown implementation and its analysis discloses the structure of the unknown program code. To quickly find code similarities, we plot the resulting similarity matrix and inspect it visually. Visual inspection is an extremely powerful tool when the unknown program contains a one-to-one copy of the original code, in which case the matrix will contain many diagonal lines with absolute correlation coefficients close to 1.

Detection Tool: Maximum Projection Rows and columns of the similarity matrix S represent the time measured in samples or clock cycles. To analyze the similarity matrix computationally, we project the matrix either onto the rows or onto the columns by using the maximum function. By projecting along the rows we can see which clock cycles of the genuine implementation are covered by the unknown implementation. Concretely, the maximum projection of S onto the rows results in an M_2-dimensional vector

$$\boldsymbol{p_{row}}^T = \left(\max_{i=1,\cdots,M_1} S_{i1}, \cdots, \max_{i=1,\cdots,M_1} S_{iM_2} \right)$$

On the other hand, projecting the matrix onto the columns will reveal the clock cycles where the genuine implementation processes the same data as the unknown one. The M_1-dimensional vector $\boldsymbol{p_{col}}$ is computed as

$$\boldsymbol{p_{col}} = \begin{pmatrix} \max_{j=1,\cdots,M_2} S_{1j} \\ \vdots \\ \max_{j=1,\cdots,M_2} S_{M_1 j} \end{pmatrix}$$

For both projection vectors it holds that sub-parts with high correlation values in succession suggest that the same intermediate data is processed using the same code segment by the microcontroller. For the quantification of similarity we use the mean absolute correlation coefficient calculated over all entries of $\boldsymbol{p_{row}}$ and $\boldsymbol{p_{col}}$

$$\rho_{row} = \frac{1}{M_2} \sum_{i=1}^{M_2} p_{row_i} \quad \text{and} \quad \rho_{col} = \frac{1}{M_1} \sum_{i=1}^{M_1} p_{col_i}.$$

Accordingly, local similarity is assessed by computing the mean absolute correlation coefficient over pre-selected sub-parts of the vectors $\boldsymbol{p_{row}}$ and $\boldsymbol{p_{col}}$.

3 Experiments

We evaluate our approach on different smartcards with ATmega163 microcontroller [8]. It is an 8-bit RISC microcontroller based on the AVR architecture running at 4MHz with 16K bytes of flash memory, 1024 bytes internal SRAM, and 32 general-purpose registers [9]. The processor uses a two-stage pipeline, where one instruction is executed while the next instruction is fetched and decoded. We measure the voltage variations of the smartcard over a resistor inserted between the ground path from the smartcard socket to the power supply using a digital oscilloscope (PicoScope 6402C) running at the sampling frequency of 325MHz. For each implementation we recorded 10.000 power traces.

We test our approach on five implementations of AES encryption written in assembly language. The AES implementations serve as an example of several implementations of the same data-dependent algorithm. Thereby we target the problem of distinguishing several implementations of the same algorithm or application. The purpose of our experiments is not only to show that identical implementations can be detected as a whole, but also which of their code segments are similar or even identical to each other.

Our test implementations differ in their overall duration and structure. An overview is shown in Figure 1. *AES-0* was written by us, *AES Labor* is available from [10], *Fast*, *Furious*,

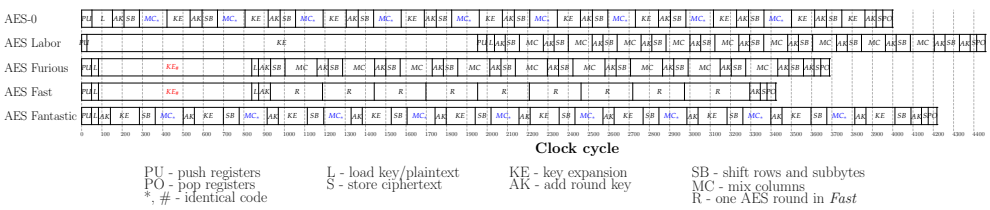

Fig. 1. AES implementations used in our experiments.

and *Fantastic* are available from [11]. *Fast* uses two sbox tables, which results in data leakage that is very different from the leakage of all other implementations. Some code parts of different implementations are the same. For example, the key expansion of *Fast* and *Furious* are identical. The MixColumns operation of *AES-0* and *AES Fantastic* follow the same low-level specification of the algorithm but utilize different registers and data addresses.

We explore two cases of plagiarism: i) The passive adversary copies the entire machine code or parts thereof into his own implementation. This case is considered in Section 3.1. ii) The active adversary modifies the machine code in various ways before copying it or parts thereof into his own implementation. This case is covered in Section 3.2.

3.1 Plagiarized Code without Modification

Visual inspection By visually inspecting a similarity matrix, we can detect identical and similar code. Figure 2 shows segments of similarity matrices for all implementations and the following AES operations: ExpandKey_1, AddRoundKey_2, ShiftRows_2 and SubBytes_2, and MixColumns_2, where subscripts denote the round of the corresponding function. When comparing each implementation to itself, we see long lines of high correlation along the diagonal in the similarity matrices, as shown in Figure 10. This is expected, since the same implementations process the same data using the same sequence of instructions.

Several observations can be made. For example, key expansion is different in all implementations except for *Fast* and *Furious*, where it is identical. AddRoundKey_2 is similar for all implementations except for *Fast*. MixColumns_2 is identical in *AES-0* and *Fantastic*, except for the registers used. The other operations are similar but not identical, and we can see that the similar values are processed at slightly different times.

Maximum projection Figure 3 shows the results of projecting the maximum values onto the rows of the similarity matrices for *Furious* and all the other tested AES implementations. When comparing *Furious* with *Furious*, we have used two different sets of traces. The maximum projection for *Furious* is the highest when it is compared to itself. The graphs also suggest that *Fast* and *Furious* implementations have identical key expansion algorithm. The high plateaus in *AES Labor* are executions of AddRoundKey. The correlation at the same times is still significant for all other implementations except for *Fast*, for which only the very first and the very last key additions shows high correlated sequences. By projecting a known implementation onto unknown implementation, we can uncover the structure of the unknown implementations, as shown in Figure 7.

Maximum projection graphs can be summarized by computing the mean absolute correlation coefficient. The resulting number indicates the similarity between the implemen-

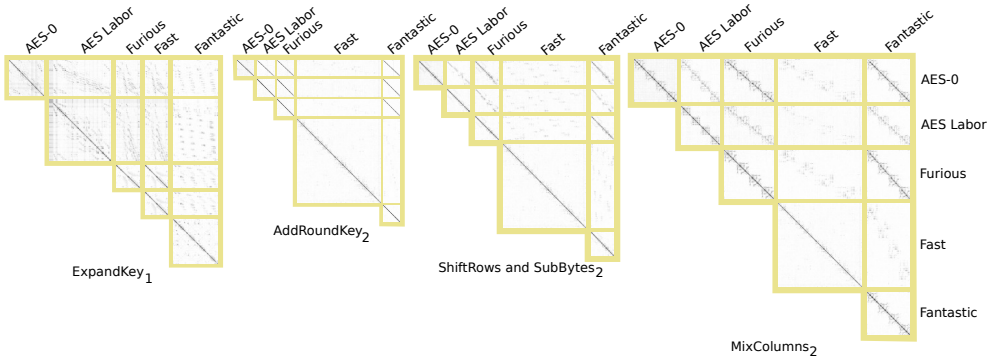

Fig. 2. Segments of similarity matrices for selected subparts of all AES implementations. Black points signify high absolute correlation (close to 1), and white points signify low correlation (close to 0). Subscripts of selected AES subparts denote the AES round: ExpandKey_1 is the key expansion in the first AES round (the key will be used by AddRoundKey_2). *RijndaelFast* has no clear distinction between AddRoundKey, ShiftRows, SubBytes, and MixColumns, so that we consider its entire second round when we compare it to the other implementations.

tations. Table 1 shows the mean absolute correlation coefficients for all our test AES implementations. Identical implementations have the mean correlation close to 1.0. Similar implementations, such as *AES-0*, *Furious*, and *Fantastic* result in high mean correlation coefficients. Even though *Fast* is very different from the other implementations, its intermediate values are similar enough to produce a significant correlation, so that we can conclude that *Fast* is performing an AES encryption. In contrast, when different data is processed, the mean absolute correlation is low, as shown in Figure 4.

Table 1. Mean absolute correlation of projecting traces of each implementation (row) into traces of each other implementation (column).

	AES-0	AES Labor	Furious	Fast	Fantastic
AES-0	**0.97**	0.41	0.63	0.33	0.53
AES Labor	0.42	**0.91**	0.46	0.29	0.39
Furious	0.61	0.44	**0.96**	0.45	0.54
Fast	0.35	0.32	0.46	**0.96**	0.29
Fantastic	0.58	0.40	0.62	0.30	**0.93**

To reveal more details about a suspicious implementation at hand, the maximum projection graph can also be summarized by computing the mean absolute correlation for each known operation, as shown in Table 2. Here we can see which subparts of the code are similar or even identical. For example, we know that the key expansion of *Fast* and *Furious* is identical; and that *AES-0* and *Fantastic* have similar *MixColumns*.

3.2 Modified Plagiarized Code

To simulate an attacker who actively manipulates plagiarized code, we have selected *Furious* as our genuine implementation and modified it in various ways, cf. [4, 12].

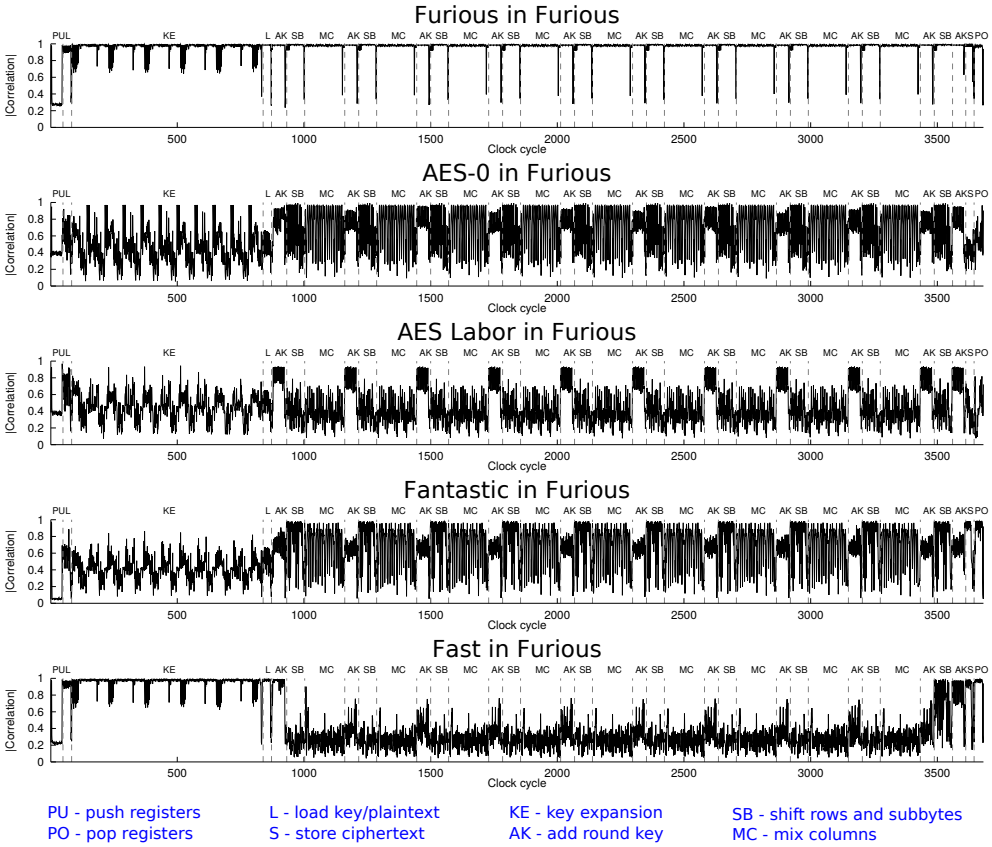

Fig. 3. Maximum projection of all AES implementations onto *Furious*.

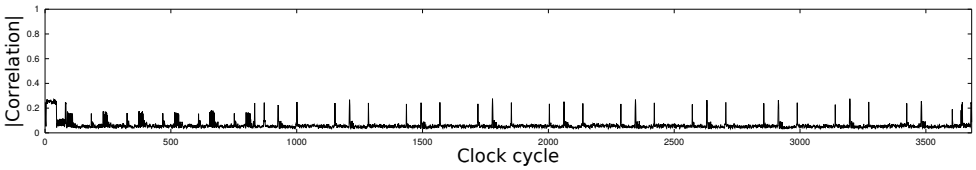

Fig. 4. Absolute correlation of *Furious* with *Furious* when non-matching data was used. Total mean absolute correlation equals to 0.061. High peaks happen when constants are processed (e.g. loading a constant into a register).

Address modification (addr) In this attack, we change the usage of all registers of the *Furious* implementation. For example, register `r8` is consistently used instead of `r0`. In addition, the variables for the key, the expanded key, the plaintext, and the ciphertext are stored in different locations in the SRAM. Finally, the constants `sbox` and `xtime` are moved to different flash addresses.

	AK	SB	MC	KE
AES-0	**0.96**	**0.97**	**0.98**	**0.97**
AES Labor	0.64	0.33	0.36	0.43
Furious	0.68	0.65	0.73	0.46
Fast	0.45	0.31	0.26	0.44
Fantastic	0.64	0.58	0.75	0.41

(a) →*AES-0*

	AK	SB	MC	KE
AES-0	0.68	0.31	0.38	0.40
AES Labor	**0.96**	**0.97**	**0.96**	**0.88**
Furious	0.73	0.38	0.40	0.41
Fast	0.48	0.24	0.19	0.39
Fantastic	0.62	0.31	0.37	0.43

(b) →*AES Labor*

	AK	SB	MC	KE
AES-0	0.71	0.65	0.71	0.46
AES Labor	0.75	0.40	0.37	0.45
Furious	**0.95**	**0.98**	**0.98**	**0.96**
Fast	0.47	0.31	0.27	**0.95**
Fantastic	0.65	0.72	0.68	0.41

(c) →*Furious*

	AK	KE	R
AES-0	0.69	0.46	0.28
AES Labor	0.73	0.45	0.23
Furious	0.85	**0.95**	0.27
Fast	**0.97**	**0.95**	**0.98**
Fantastic	0.64	0.40	0.25

(d) →*Fast*

	AK	SB	MC	KE
AES-0	0.66	0.57	0.75	0.33
AES Labor	0.62	0.32	0.35	0.40
Furious	0.62	0.71	0.70	0.32
Fast	0.43	0.27	0.25	0.31
Fantastic	**0.96**	**0.96**	**0.97**	**0.90**

(e) →*Fantastic*

Table 2. Mean absolute correlation of projecting similarity matrices of all AES implementations into the implementations denoted by symbol "→". Numbers in bold signify identical code. AK-*AddRoundKey*, SB-*SubBytes*, MC-*MixColumns*, KE-*KeyExpansion*.

Instruction reordering (swap) In this attack, we change the order of individual instructions, and also swap entire blocks of instruction without changing the number of clock cycles required to perform the encryption. Concretely, we change the order of loading and saving the key, plaintext, and ciphertext; the order of applying `XOR` in AddRoundKey; the order of rows when computing ShiftRows and SubBytes; and the order of applying `XORs` in the key scheduler. On several occasions, in order to change the sequence of loading variables, we change instructions such as, e.g., `LD R0, Z+` (opcode 9001) into instructions like `LDD R0, Z+0` (opcode 8000).

Combination of addr and swap (addr+swap) Here, we combine the address changes and the swapping of the instructions.

Insert dummy NOPs (dummy) This version has `NOP` instructions inserted throughout the code: in the key expansion, in SubBytes, ShiftRows, and MixColumns. As a result, the encryption needs 792 additional clock cycles to finish.

Insert other dummy instructions (dummy smart) The problem with `NOP` instructions is that on the ATmega163 they can be easily distinguished because they have low power consumption in comparison to the other instructions. To make the power consumption appear more genuine, we insert dummy instructions that manipulate the state for a short amount of time, and change it back before resuming with encryption. Figure 8 shows several assembly macros that we spread throughout the code that reuse intermediate values of the implementation. This version also has 792 additional clock cycles.

All attacks combined (dummy smart+addr+swap) This attack introduces 792 smart dummy cycles and combines them with the add+swap attack.

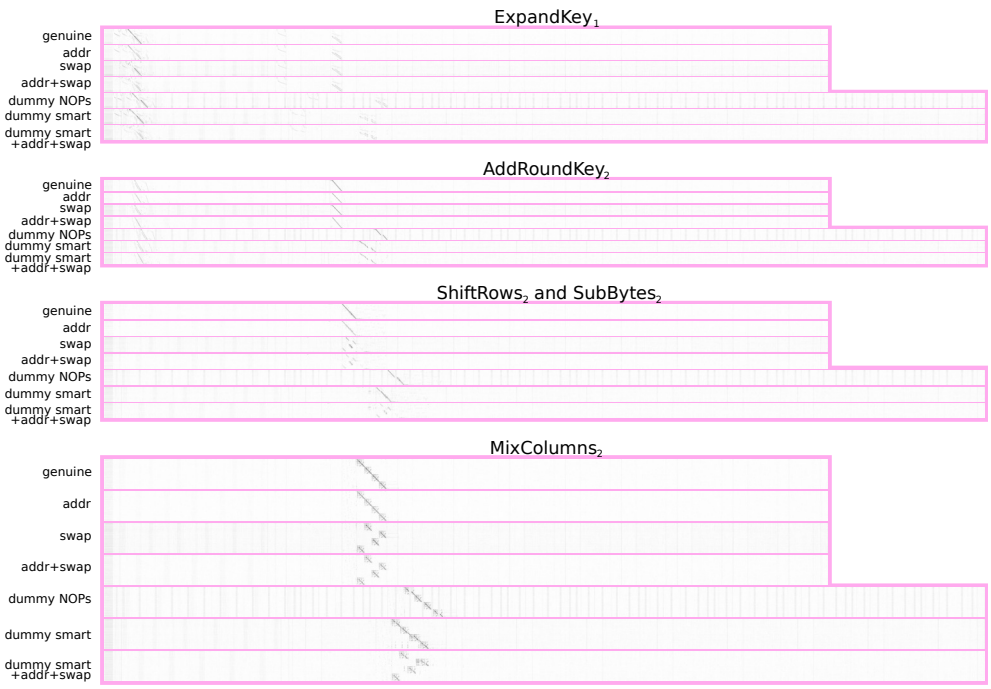

Fig. 5. Similarity matrices of plagiarism attacks on RijndaelFurious (denoted by "original"). X and Y axes are measured in clock cycles.

Visual inspection Figure 5 shows excerpts of the similarity matrices that correspond to the genuine implementation along the Y-axis, and the manipulated implementations along the X-axis for selected AES functions. Figures 11 to 16 show the full similarity matrices. Code with manipulations that do not change the order of instructions can be well recognized because of the long lines along the diagonals of the matrices.

Maximum projection Figures 6 and 9 show the maximum projection method applied on similarity matrices of *Furious* with all the other implementations. The versions *addr* and *swap* produce the largest impact on the maximum projection method, however, we are still able to distinguish MixColumns quite well. On the other hand, despite introducing 792 dummy clock cycles, we can argue with a high confidence that the presented system contains large portions of our genuine implementation.

Table 3 shows the total mean absolute correlation coefficients and the operation-wise absolute mean correlation coefficient for *Furious* and its modified versions. Here we can see that our method is well-suited for finding versions of our code with added dummy instructions—even 792 additional dummy clock cycles have little effect on the mean absolute correlation, which, at >0.8 is high enough to suspect a plagiarism. On the other hand, by exchanging registers and choosing different addresses for variables and constants in the memory, as in *addr*, the attacker can decrease the mean absolute correlation by a large margin. However, after cross-referencing the similarity matrix, as shown in Figure 11, we see an almost continuous line along its diagonal, implying an identical implementation. Swapping

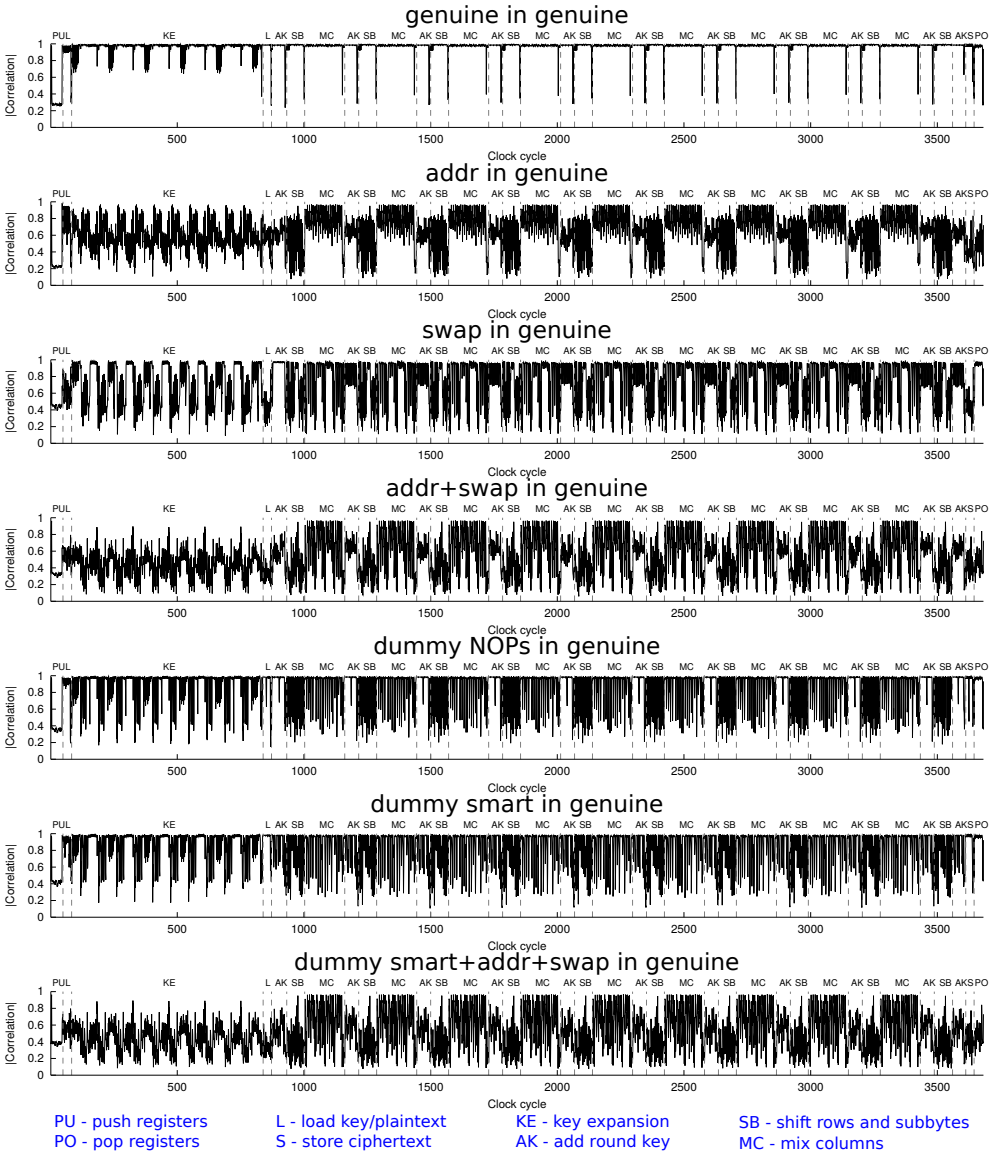

Fig. 6. Maximum projection of modified implementations onto genuine *Furious*.

code chunks does not help much, since the individual parts are still well-recognizable in the operation-wise mean correlation table and in the similarity matrix. Thus, in order to successfully avoid raising suspicion, an attacker will have to use a plethora of countermeasures at a much finer level, which requires a considerable amount of effort to obfuscate the code without introducing bugs and unintended side effects.

	genuine
genuine	**0.96**
addr	0.64
swap	0.73
addr+swap	0.52
dummy NOPs	0.84
dummy smart	0.83
dummy smart+addr+swap	0.51

(a)

	AK	SB	MC	KE
genuine	**0.95**	**0.98**	**0.98**	**0.96**
addr	0.61	0.52	0.76	0.60
swap	0.84	0.62	0.78	0.80
addr+swap	0.59	0.37	0.64	0.45
dummy NOPs	0.92	0.72	0.87	0.86
dummy smart	0.82	0.75	0.85	0.85
dummy smart+addr+swap	0.54	0.36	0.63	0.44

(b)

Table 3. (a) Total mean absolute correlation of modified *Furious* implementations projected into the genuine *Furious* implementation. (b) Operation-wise mean absolute correlation of modified *Furious* implementations projected into the genuine *Furious* implementation.

4 Discussion

Our work is built upon the assumption that identical code produces almost identical side channel leakage on an identical (but physically distinct) microcontroller platform. In our experiments we studied five different AES implementations and observed significant correlation coefficients in each similarity matrix. This is due to the fact that all test programs are implementations of the same task and process the same data. However, as exemplarily shown in Fig. 3, the maximum projection of identical implementations clearly stands out when compared to different implementations, and thereby underlines that the same sequence of instructions is clearly highlighted and allows us to detect plagiarized software of a passive adversary with high accuracy.

In the case when an adversary has spent some effort modifying the original code, we come to the conclusion that our method achieves a good robustness against changed registers and added dummy cycles. The second case is a clear advantage compared to the method of Durvaux et al. [4] that turned out to be very sensitive to the addition of 57 dummy clock cycles, whereas we worked with 792 additional dummy cycles. In addition, in contrast to the method of Durvaux et al., we are not forced to cut the recorded traces to have the same length and are able to use all recorded information. The robustness of our approach to adding smart dummy instructions is only slightly worse compared to dummy NOPs and for many code segments the original sequence is still revealed in the maximum projection. Similar observations hold when instructions are swapped.

In this paper we have used static, time-aligned, and constant-time applications to test our approach. However, we are confident that our methods are also applicable in a more general setting, e.g., with code branches and nondeterministic parts. A further adversary strategy could be to implement dynamic code generation on the microcontroller as, e.g., proposed by [12] which will make our approach more difficult if the frequency of run-time code generation is high. As result, an adversary is forced to combine many kinds of modifications and put a considerable amount of effort in order to reduce successful plagiarism detection that usually lead to enhanced code complexity and execution time.

Further promising properties of our approach can be seen in Fig. 4. Even if the data of the two implementations do not match, our approach reveals smaller but still significant correlation signals. These correlation signals are assumed to be due to code similarities and lead to the conjecture that our methods are still able to detect the similarity of two programs

if there is a lack of any input data channel. Interestingly, this observation gives reasons to assume that even the implementation of an intrinsic data masking scheme by the adversary might be not sufficient.

Thinking from a higher-level perspective, our tools detect whether a program processes identical data or not. Hereby, patent infringements on the algorithm level may be possible to detect. For example, an unlicensed use of a patented algorithm with a specific data processing can be proven using the data dependent side-channel leakage based on our approach.

5 Conclusions

We have presented and evaluated new methods based on characteristic data leakage for detecting software plagiarism on a microcontroller platform. The conducted experiments give evidence that these methods are highly robust to many different code transformations and that the resulting pattern of data leakage of the microcontroller executing the code can be considered as its own watermark. Promising research directions are opened for connecting horizontal and vertical dimensions for code sequence analysis through side channel traces.

Acknowledgement 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. G. T. Becker, W. Burleson, and C. Paar, "Side-channel watermarks for embedded software," in *9th IEEE NEWCAS Conference (NEWCAS 2011)*, 2011.
2. G. Becker, D. Strobel, C. Paar, and W. Burleson, "Detecting software theft in embedded systems: A side-channel approach," *Information Forensics and Security, IEEE Transactions on*, vol. 7, no. 4, pp. 1144–1154, 2012.
3. D. Strobel, F. Bache, D. Oswald, F. Schellenberg, and C. Paar, "SCANDALee: A Side-ChANnel-based DisAssembLer using Local Electromagnetic Emanations," in *Design, Automation, and Test in Europe (DATE)*, March 9–13 2015.
4. F. Durvaux, B. Gérard, S. Kerckhof, F. Koeune, and F.-X. Standaert, "Intellectual property protection for integrated systems using soft physical hash functions," in *Information Security Applications* (D. Lee and M. Yung, eds.), vol. 7690 of *LNCS*, pp. 208–225, Springer Berlin Heidelberg, 2012.
5. P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings* (M. Wiener, ed.), (Berlin, Heidelberg), pp. 388–397, Springer Berlin Heidelberg, 1999.
6. S. Kerckhof, F. Durvaux, F.-X. Standaert, and B. Gerard, "Intellectual property protection for FPGA designs with soft physical hash functions: First experimental results," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pp. 7–12, June 2013.
7. S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks Revealing the Secrets of Smart Cards*. Springer US, 2007.
8. Atmel, *ATmega163(L) Datasheet (revision E)*, Feb. 2003.
9. Atmel, *Atmel AVR 8-bit Instruction Set Manual (revision 0856J)*, July 2014.
10. D. Otte, "Avr-crypto-lib." https://www.das-labor.org/wiki/AVR-Crypto-Lib/en.
11. B. Poettering, "AVRAES: The AES block cipher on AVR controllers."
12. D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet, "Runtime code polymorphism as a protection against side channel attacks," in *Information Security Theory and Practice: 10th IFIP WG 11.2 International Conference, WISTP 2016, Heraklion, Crete, Greece, September 26–27, 2016, Proceedings* (S. Foresti and J. Lopez, eds.), pp. 136–152, Cham: Springer International Publishing, 2016.

6 Appendix

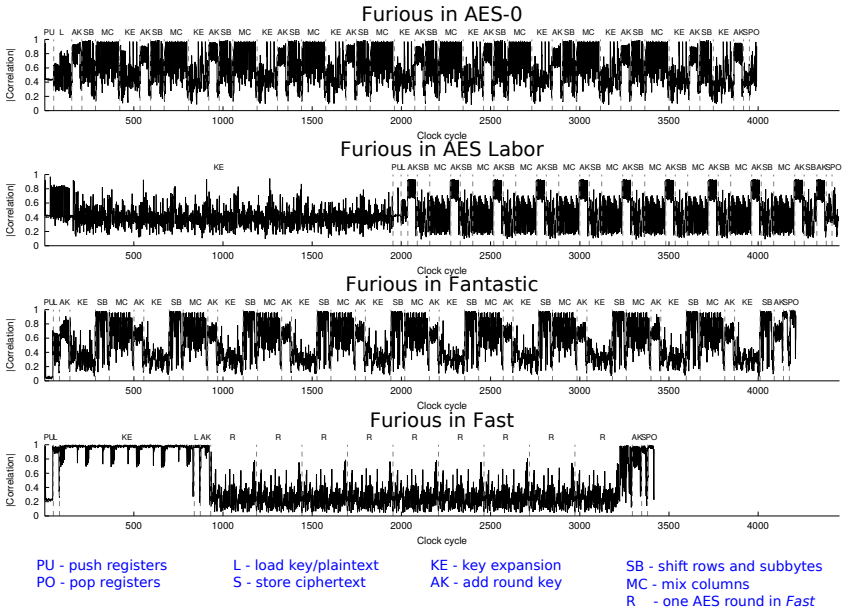

Fig. 7. Maximum projection of *Furious* onto all other AES implementations.

① ② ③

```
INC \reg
DEC \reg
```

```
NEG \reg
NEG \reg
```

```
ROL \reg
ROR \reg
```

④

```
PUSH \tmp
LDI  \tmp, \c
EOR  \reg, \tmp
POP  \tmp
```

⑤

```
LDI ZL, 0x00
LPM \tmp, Z
```

⑥

```
EOR \tmp, \tmp
```

⑦

```
PUSH \reg1
PUSH \reg2
PUSH \reg3
EOR  \reg1, \reg2
EOR  \reg2, \reg3
EOR  \reg3, \reg1
POP  \reg3
POP  \reg2
POP  \reg1
```

⑧

```
MOV \tmp, \reg ;; save register
LDI ZH, hi8(hd_temp)
LDI ZL, lo8(hd_temp)
LD  \reg, z
MOV \reg, \tmp ;; restore register
```

Fig. 8. Assembly macros used to insert dummy smart instructions. Macros 1,2,3 change the content of a chosen register in one clock cycle, and change it back in the next one. Macro 8 is used to remove Hamming-distance leakage between consecutive SRAM reads or writes by performing a dummy read in the SRAM at some constant address. Macro 5 is used before some of the sbox lookups in the flash memory. Macro 6 is applied to an unused register and leaks data from preceding operations that have use the ALU (arithmetic-logic unit). Macro 4 loads a random constant value chosen at compile time into a register and restores the register right after that. Macro 7 uses XORs on three selected registers and immediately restores them to their respective original values.

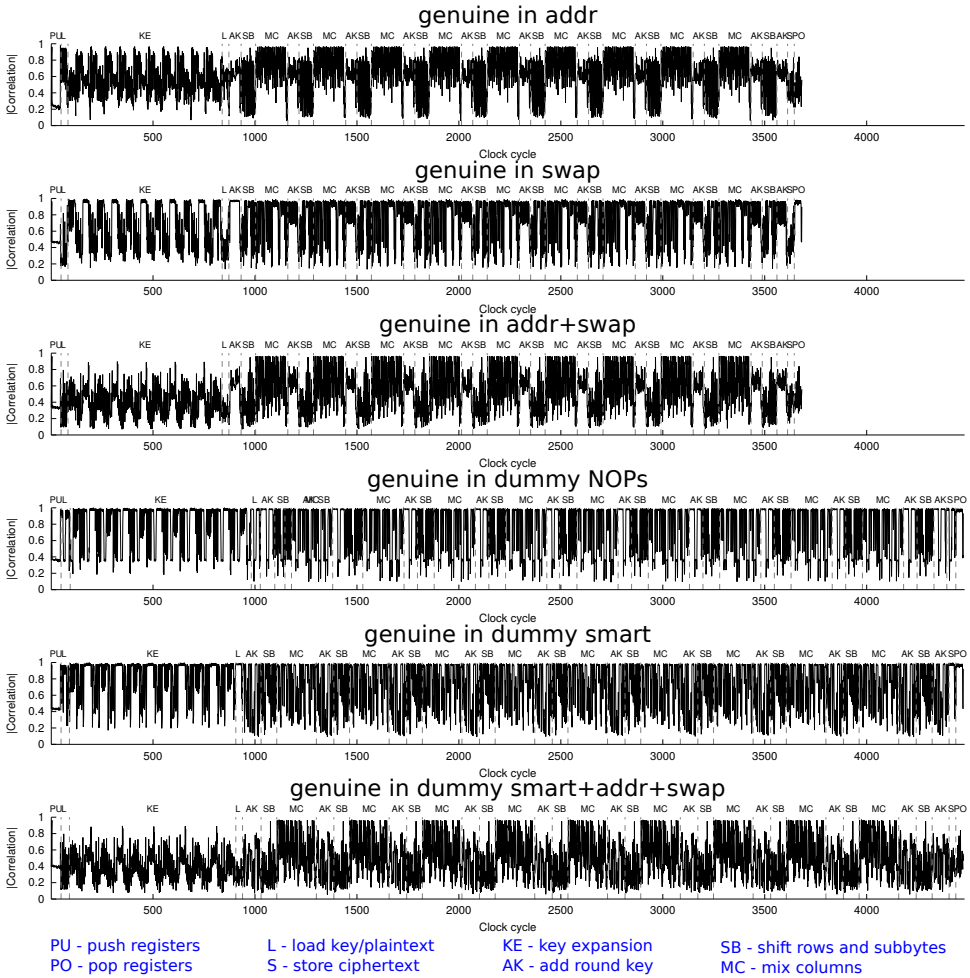

Fig. 9. Maximum projection of genuine *Furious* onto all modified AES implementations.

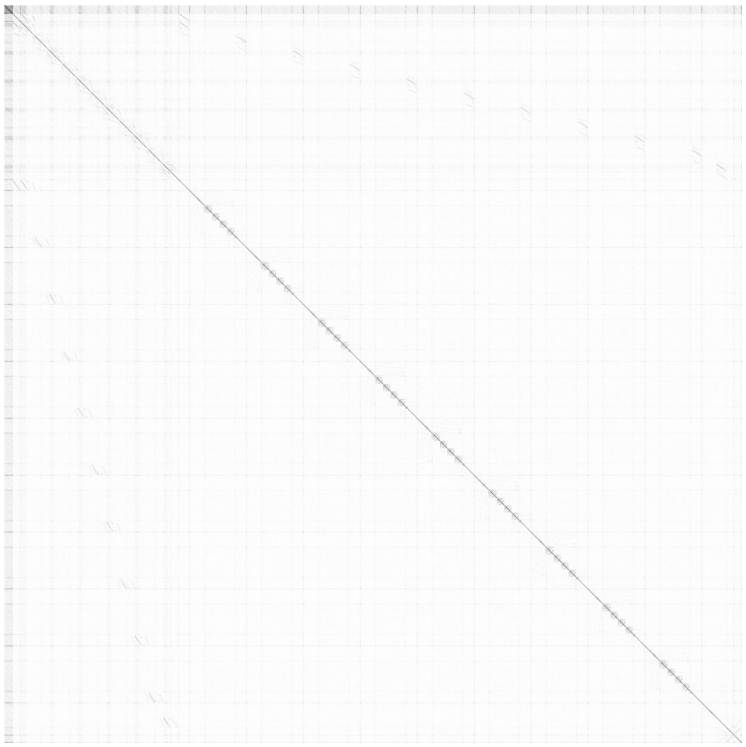

Fig. 10. Similarity matrix of *Furious* with itself.

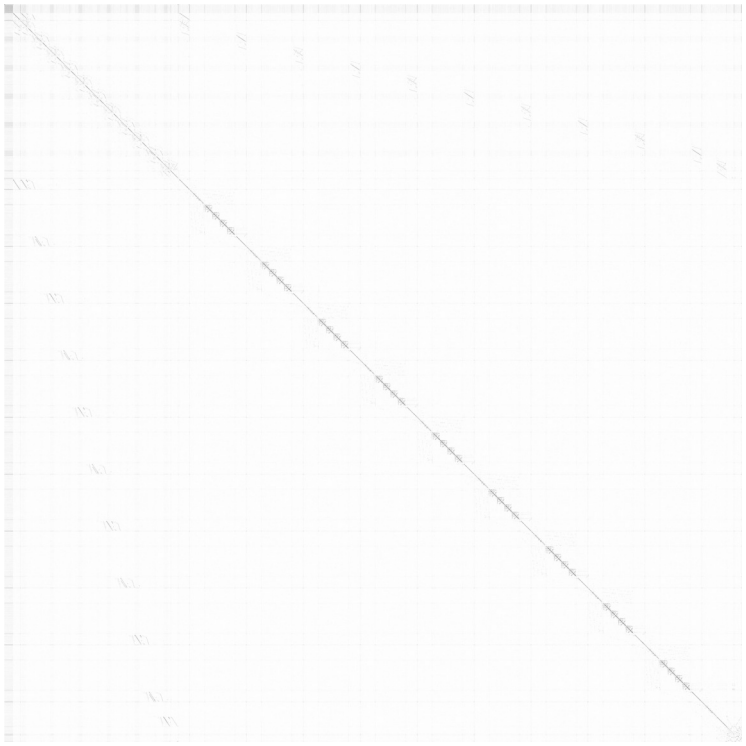

Fig. 11. Similarity matrix of *addr* and the genuine *Furious* AES implementations.

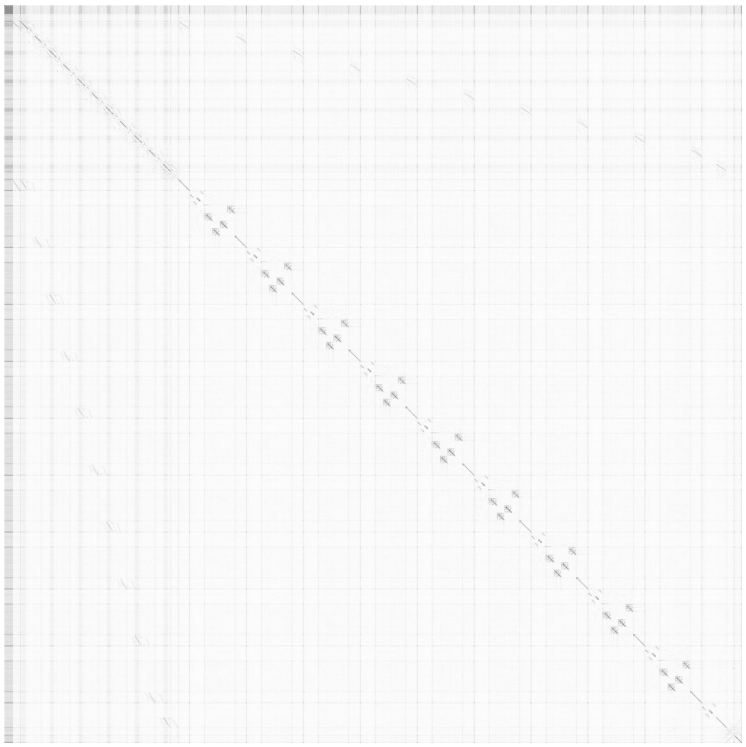

Fig. 12. Similarity matrix of *swap* and the genuine *Furious* AES implementations.

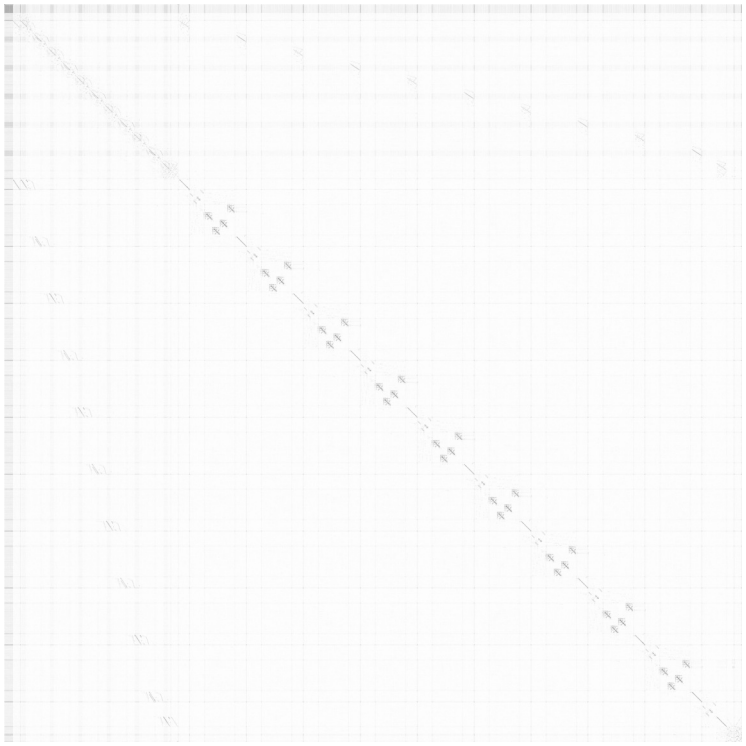

Fig. 13. Similarity matrix of *addr+swap* and the genuine *Furious* AES implementations.

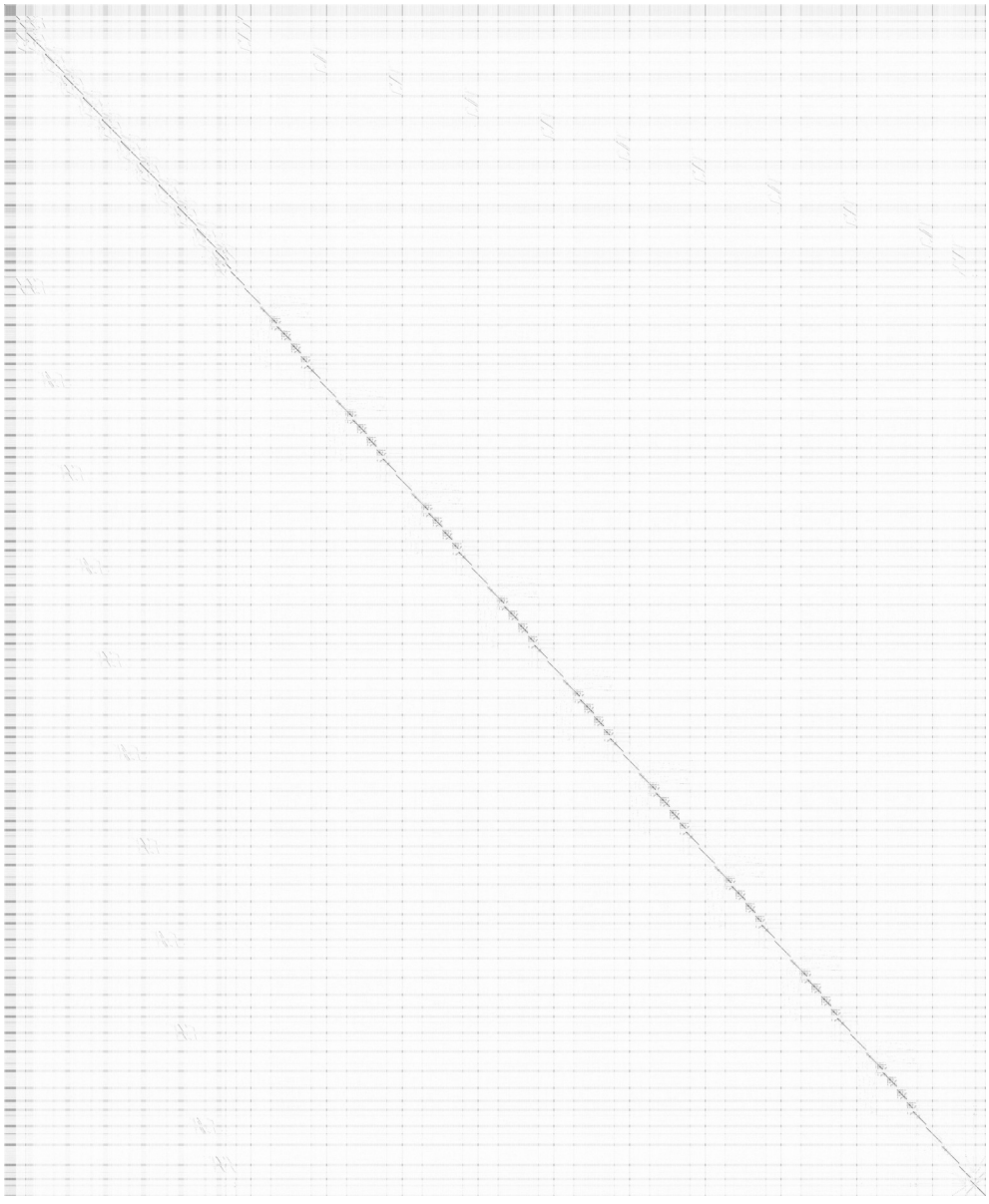

Fig. 14. Similarity matrix of *dummy NOPs* and the genuine *Furious* AES implementations.

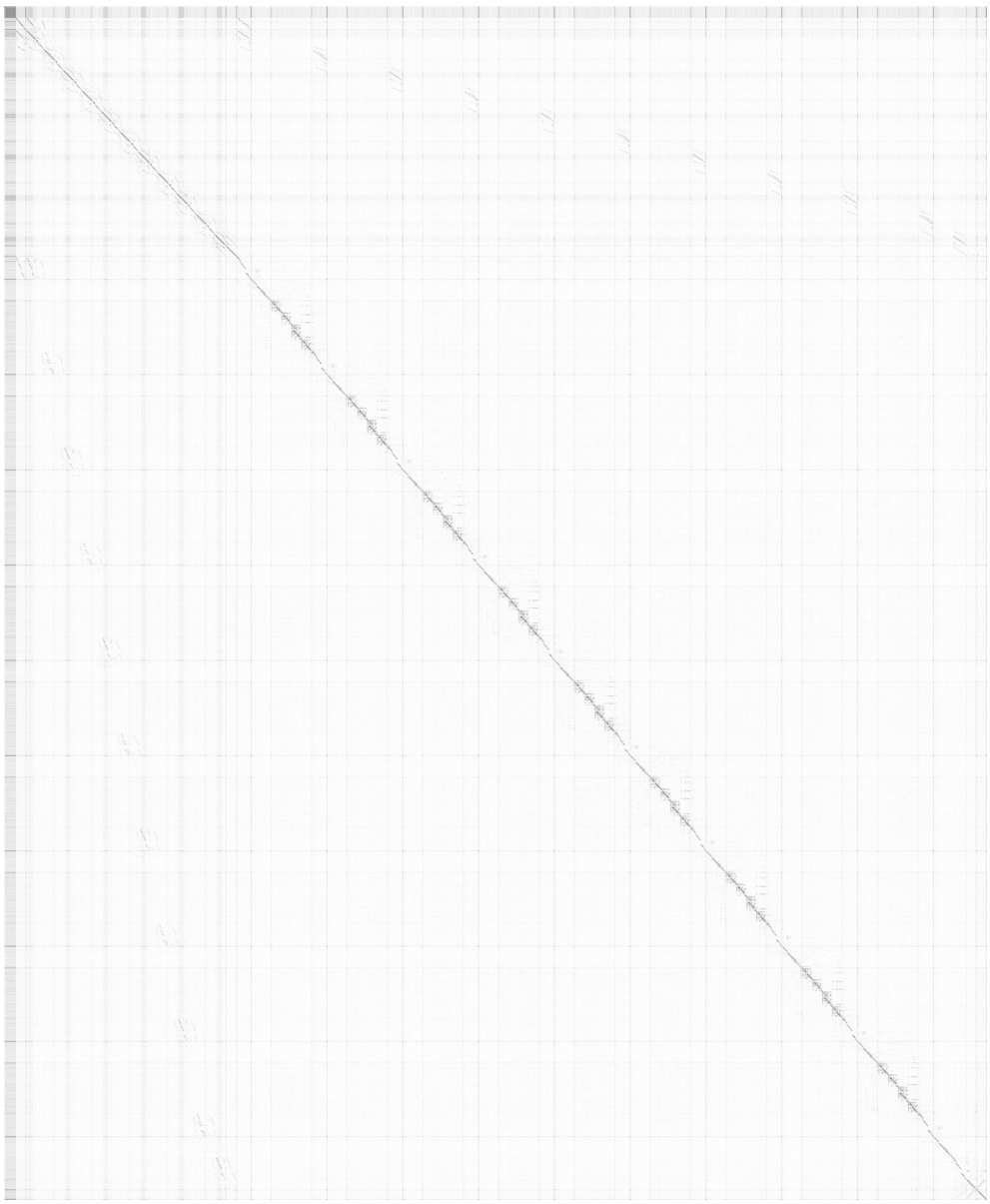

Fig. 15. Similarity matrix of *dummy smart* and the genuine *Furious* AES implementations.

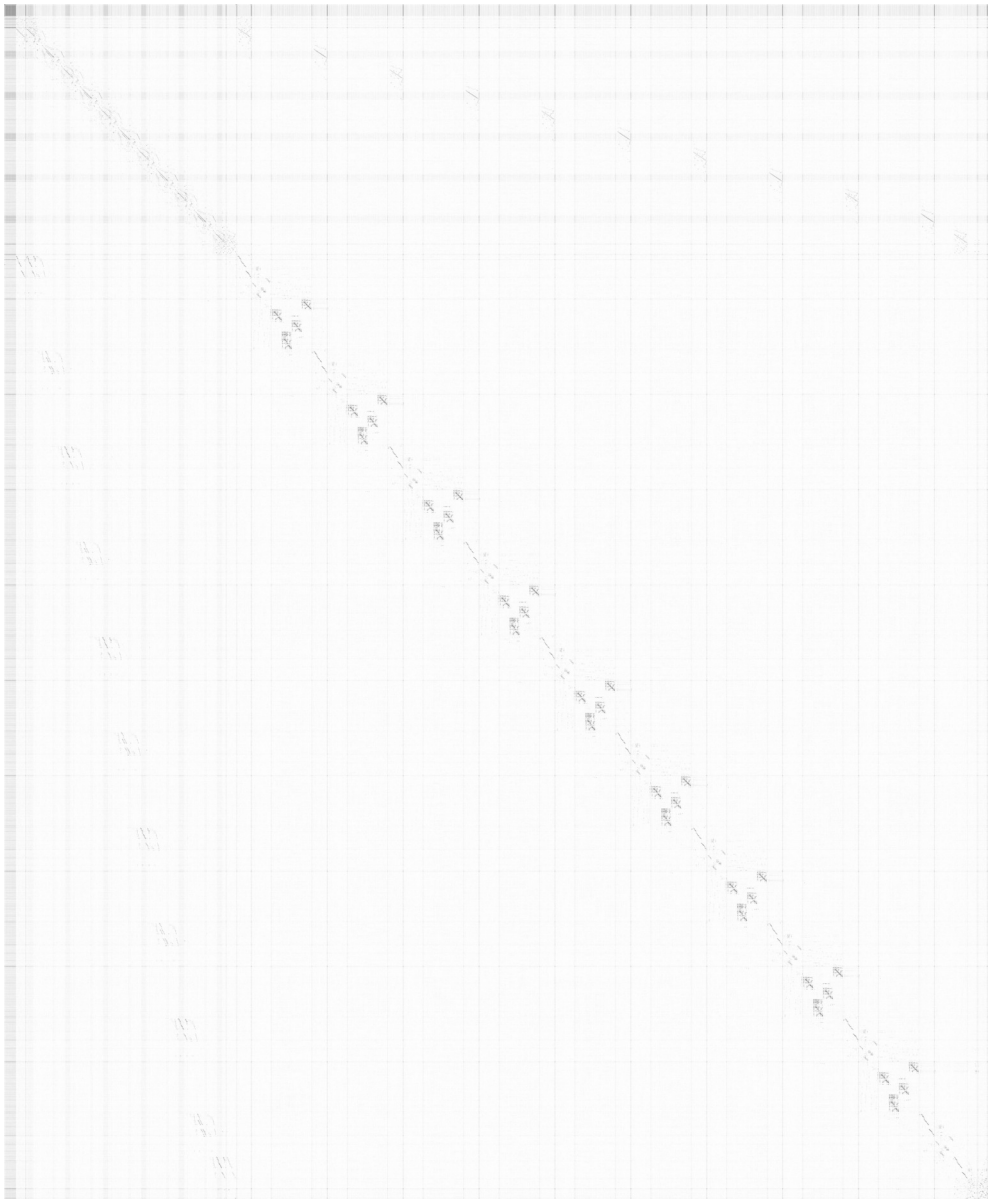

Fig. 16. Similarity matrix of *dummy smart+addr+swap* and the genuine *Furious* AES implementations.