Detecting Software Theft in Embedded Systems: A Side-Channel Approach

Georg T. Becker, Daehyun Strobel, Christof Paar, Fellow, IEEE, Wayne Burleson, Fellow, IEEE

Abstract—Source code plagiarism has become a serious problem for the industry. Although there exist many software solutions for comparing source codes, they are often not practical in the embedded environment. Today’s microcontrollers have frequently implemented a memory read protection that prevents a verifier from reading out the necessary source code. In this paper, we present three verification methods to detect software plagiarism in embedded software without knowing the implemented source code. All three approaches make use of side-channel information that is obtained during the execution of the suspicious code. The first method is passive, i.e., no previous modification of the original code is required. It determines the Hamming weights of the executed instructions of the suspicious device and uses string matching algorithms for comparisons with a reference implementation. In contrast, the second method inserts additional code fragments as a watermark that can be identified in the power consumption of the executed source code. As a third method, we present how this watermark can be extended by using a signature that serves as a proof-of-ownership. We show that particularly the last two approaches are very robust against code-transformation attacks.

Index Terms—Side-channel analysis, software watermarking, IP protection, embedded systems.

I. INTRODUCTION

Software plagiarism and piracy is a serious problem which is estimated to cost the software industry billions of dollars per year [6]. Software piracy for desktop computers has gained most of the attention in the past. However, software plagiarism and software piracy is also a huge problem for companies working with embedded systems. In this paper, we focus on the unique challenge to detect software plagiarism and piracy in embedded systems. If a designer suspects that his code has been used in an embedded system, it is quite complicated for her to determine whether or not her suspicion is true. Usually she needs to have access to the program code of the suspected device to be able to compare the code with the original code. However, program memory protection mechanisms that prevent unauthorized read access to the program memory are frequently used in today’s microcontrollers. Hence, to gain access to the program code of an embedded system these protection mechanisms would have to be defeated first. This makes testing embedded devices towards software plagiarism very difficult, especially if it needs to be done in an automated way. Furthermore, the person who is illegally using unlicensed code might apply code-transformation techniques to hide the fact that he is using someone else’s code. In this case detecting the software theft is hard even if the program code is known.

Software watermarks [23] enable a verifier to test and prove the ownership of a piece of software code. Previously proposed watermarks usually require access to the program code [11] or the data memory [22], [7] during the execution of the program to detect the watermark. But as mentioned before, access to the memory and the program code is usually restricted in the embedded environment. Most software watermarks proposed in the literature are written in C or assembly. Furthermore, many previously proposed software watermarks have shown to be not very robust to code-transformations [11].

In this paper, we introduce new and very efficient methods to detect software theft in embedded systems. To determine whether or not an embedded system is using our code, we use side-channel information that the device leaks out while the suspected code is running on the device. In this way, we avoid the need to gain access to the program or data memory. The basic idea is to measure the power consumption of the device under test and use these measurements to decide whether or not a piece of unlicensed code is running on the device. We present three different levels of protection. The first method is used to detect software thefts that are copies of the original software code with only minor changes. From the power measurements we are able to derive the Hamming weight of each executed assembler instruction. Using string matching algorithms we can then efficiently detect whether a suspected code is equal or similar to the reference code. The great advantage of this method is that no changes to the original code need to be done to be able to detect the code, i.e., no watermark needs to be added. Hence, this method does not introduce any kind of overhead and can be applied to any existing code.

By using code-transformations (see Sect. V-B) an attacker might be able to modify the code in a way that it is still functioning correctly but cannot be detected
using the above mentioned method. For higher protection against these types of attacks we therefore propose a side-channel software watermark. The side-channel software watermark consists of only a few instructions that are inserted at the assembly level. These instructions are then leaking information in the power consumption of the tested design and can be detected by a trusted verifier with methods very similar to classic side-channel attacks. These watermarks provide higher robustness against code-transformation attacks while introducing only a very small overhead in terms of performance and code size.

The previous two methods to detect software plagiarism can be used by a verifier to detect whether or not his code is present in an embedded system. However, these methods cannot be used to prove towards a third party that the code or the watermark belongs to the verifier. The original side-channel software watermark only transmits one bit of information – either the watermark is present or not – but it does not contain any information about the owner of the watermark. We therefore extended the side-channel watermark so that the watermark can transmit a digital signature. The verifier can use this digital signature to prove towards a third party, e.g., a judge, that he is the owner of the watermark. In most cases this will imply that he is also the owner of part of the software code that contained this watermark.

The rest of the paper is organized as follows. The next section explains in detail how the detection based on Hamming weights work. In Sect. III the software side-channel watermark is introduced. How this watermark idea can be extended to provide proof-of-ownership is presented in Sect. IV. We will then discuss the robustness of the proposed side-channel software watermark in Sect. V. We conclude the paper with a short summary of the achieved results in the last section.

II. Plagiarism Detection with String Matching Algorithms

Our first approach is intended to provide an indication of software plagiarism in the case where the insertion of copy detection methods inside the code is not available, e.g., for already existing products.

Let us begin with a definition of the Hamming weight of a number $D$ as this is a crucial point for this approach. Assume that $D$ is given by its $m$-bit representation $D = d_{m-1} \ldots d_1 d_0$. The Hamming weight $HW(D)$ is then defined by the number of 1’s, i.e.,

$$HW(D) = \sum_{i=0}^{m-1} d_i.$$ 

We now propose an efficient method that makes use of the fact that the power consumption of the microcontroller is related to the Hamming weight of the prefetched opcode. This dependency allows us to map the power consumption of every clock cycle, which may consist of thousands of sampling points, to only one Hamming weight. The result is a high dimensionality reduction of the power traces. For subsequent analyses, string matching algorithms are applied for comparisons between original and suspicious software executions.

Our target device is the 8-bit microcontroller Atmel AVR ATmega8. The ATmega8 is an extremely widely used microcontroller which is based on the RISC design strategy. As most of the RISC microcontrollers, the ATmega8 uses a Harvard architecture, i.e., program and data memory are physically separated and do not share one bus. The main advantage of this architecture is a pipelining concept that allows a simultaneous opcode prefetch during an instruction execution (see Fig. 1).

![Fig. 1. Pipelining concept of the ATmega8 [2]. While one instruction is executed, another one is prefetched from the program memory.](image)

The opcodes of the ATmega8 have a 16-bit format and consist of the bit representation of the instruction and corresponding operands or literals. The instruction AND, e.g., performs a logical AND between the two registers $r$ and $d$, $0 \leq d, r \leq 31$, and is given by the opcode

$$0010 \ 00r_4d_4 \ d_3d_2d_1d_0 \ r_3r_2r_1r_0,$$

where $d_4 \ldots d_0$ and $r_4 \ldots r_0$ are the binary representations of $d$ and $r$, respectively.

A closer look at the power consumption reveals that the prefetching mechanism is leaking the Hamming weight of the opcode. Figure 2 shows the power consumption of random instructions with different Hamming weights of the prefetched opcode recorded at a clock rate of 1MHz. Right before the rising to the second peak, the Hamming weights of the prefetched opcodes are clearly distinguishable from each other.

Our prediction was verified by the analysis of an AES implementation on the ATmega8. The histogram in Fig. 3 depicts the distribution of the power consumption at the point of leakage using about 6 million instructions of multiple AES encryptions. The intersections between the peaks were taken as threshold values when mapping the voltages to Hamming weights. In this way the power traces are converted to strings of Hamming weights which means that every clock cycle is represented by only one value. In our tests, the probability of a correct mapping to the Hamming weight was close to 100% and we were able to reduce 1. The only exceptions are four (of 130) instructions that have access to the program or data memory. These instructions can be described either by 16 or 32-bit opcodes. 32-bit opcodes are fetched in two clock cycles.

2. We focused on the ATmega8 here. However, we observed the same leakage behavior for other microcontrollers of the AVR family by Atmel and PIC family by Microchip. Both families use the Harvard architecture with separated buses for data and instruction memory.
The Hamming weight strings can now be used to indicate software plagiarism. The idea can be summarized in three steps: First, the execution flow of the original software is mapped to a string s of Hamming weights. This is realized either by measuring and evaluating the power consumption of the execution as stated above or by simulating the execution flow and calculating the Hamming weights from the opcodes. Note that, e.g., branch instructions need more than one clock cycle and hence, during the execution more than one opcode is prefetched from the program memory.

The second step is to record a power trace of the suspicious device to obtain also a string of Hamming weights, denoted as t. A one-to-one copy can now be detected in a third step by comparing the two strings s and t.

In fact, a one-to-one copy is an idealized assumption of software plagiarism and is therefore easy to detect. In the following we discuss how minor modifications can also be detected by the verifier.

1) Register substitution: The easiest modification that influences the Hamming weights is to substitute the working registers. There are 32 general purpose working registers on the ATmega8, which means that one can choose between six different Hamming weights (register $R1 \rightarrow 00000$ to register $R31 \rightarrow 11111$). However, a substitution of a register affects the Hamming weight of every instruction that processes the data of the changed register. For instance, the short assembler code

LDI R16,0x34 ;loads the constant directly
LDI R17,0x53 ;to the given register
ADD R16,R17 ;adds the two registers and
            ;places the result in R16

results to the string $s = (6,8,5)$ (cf. [1]). If the attacker substitutes, e.g., R16 with R23, this affects line 1 and 3 and leads to $t = (9,8,8)$. To consider register substitutions, the verification process can now be extended by a generalized string $s'$ that considers the dependencies between the registers in different instructions. For instance, the string of our short example could be changed to $s' = (6+x,7+y,2+x+y)$, where 6, 7 and 2 are the minimum Hamming weights of the given instructions excluding the registers and $x$ and $y$ are variables for the Hamming weights of the registers. The verifier has now to check if there exist $x$ and $y$ with $0 \leq x, y \leq 5$ such that $s'$ and $t$ match. Of course, a long string with much dependencies is more meaningful than a short string as given in our example.

2) Partial copy: Instead of scanning the whole source code, it might make sense to focus only on the important parts of the program to detect partial copies of the original code. In this case, approximate string matching algorithms like the generalized Boyer-Moore-Horspool algorithm [19] are well-suited to find all positions where these patterns occur in the string $t$. The difference to an exact string matching algorithm is that the generalized Boyer-Moore-Horspool algorithm considers a predefined number $k$ of mismatches. This is necessary due to infrequent erroneous mappings of power consumptions to Hamming weights or to consider the usage of LPM instructions. The generalized Boyer-Moore-Horspool algorithm solves the $k$ mismatches problem in $O(nk(\frac{1}{m} + \frac{1}{m^2}))$ on average, with $n$ being the length of the string, $m$ the length of the pattern and $\Sigma$ the alphabet. Additionally, a preprocessing is executed once for every pattern in $O(m + k|\Sigma|)$.

3) Insert, delete or substitute instructions: A metric that is often used for approximate string matching is the Levenshtein distance of two strings [12], also known as edit
Fig. 4. Computation of the Levenshtein distance between the two strings \( s = 1, 2, 6, 6 \) and \( t = 5, 2, 6, 5, 6 \). The gray boxes illustrate the trace that is followed to get to the highlighted overall distance 2.

distance. It was introduced by V. Levenshtein in 1965 and is defined as the minimum number of necessary edits to transform one string into another, in our case \( s \) into \( t \). Possible edits are:

- **insertion** – add one character of \( t \) to the string \( s \),
- **deletion** – remove one character of the string \( s \),
- **substitution** – replace a character of \( s \) by a character of \( t \).

Let us denote the length of \( s \) as \( m \) and the length of \( t \) as \( n \). Then the Levenshtein distance is determined by creating an \((m+1) \times (n+1)\) matrix \( C \) with the following steps:

i. Initialize the first row and column:

\[
C_{i,0} = i \quad \text{for all} \quad 0 \leq i \leq m, \\
C_{0,j} = j \quad \text{for all} \quad 1 \leq j \leq n.
\]

ii. For all \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \) set

\[
C_{i,j} = C_{i-1,j-1}
\]

if no edit is necessary, i.e., \( s_i = t_j \) with \( s_i \) and \( t_j \) denoting the \( i \)-th character of \( s \) and the \( j \)-th character of \( t \), respectively, and

\[
C_{i,j} = \begin{cases} 
C_{i-1,j-1} + 1 & \text{(substitution)} \\
C_{i,j-1} + 1 & \text{(insertion)} \\
C_{i-1,j} + 1 & \text{(deletion)}
\end{cases}
\]

else.

The overall distance between \( s \) and \( t \) is the value stored in \( C_{m,n} \). An example is given in Fig. 4. Since every cell of the matrix \( C \) has to be computed, the computational complexity is bound by \( O(mn) \).

With this method the verifier is able to measure the degree of modification or simply use the distance as an indication for plagiarism.

4) **Handling unpredictable branches or interrupts:** In some cases, it is impossible to acquire measurements of the suspicious implementation with the same input data as the reference implementation. Another scenario is that the underlying algorithm is non-deterministic regarding the order of executed subroutines due to random parameters. The result might be that for every measurement a different path of subroutines is executed. While scanning the whole string after a partial match, e.g., a subroutine, could be still successful, comparing the whole strings seems to be more complicated.

One technique that can help us here is called dot plot and comes from the field of bioinformatics where it is mostly applied on DNA sequences to illustrate similarities [10]. In [21], Yankov et al. adopted this technique to analyze two time series after converting them into a symbolic representation. The idea is very simple: to compare two strings \( s \) with length \( m \) and \( t \) with length \( n \), an \( m \times n \)-matrix is generated where position \((i,j)\) is marked if \( s_i \) and \( t_j \) match. A small example is given in Fig. 5.

Two interesting observations can be made from this plot. First, equal substrings are represented as diagonal lines. In our example the substring \((1,5,2)\) appears in both strings. The second observation is that a mirrored substring is marked as a diagonal line which is perpendicular to the main diagonal. This is the case for \((2,8,6)\) and \((6,8,2)\).

We applied this technique to two Hamming weight strings of different software implementations with both containing two equal subroutines. A small detail is depicted in Fig. 6(a).

As there is much noise in this plot, we removed every matching that is shorter than four characters. The result is given in Fig. 6(b), where at least two diagonal lines are suspicious. The first, starting at about character 60 in HW string 2 and the second, starting at about 240. Another interesting point is that the second line is interrupted. This is due to the insertion of some dummy instructions during the subroutine of the second implementation which should simulate an interrupt service routine.
The methods proposed in this section provide only an indication of plagiarism. To make a more precise statement, the software code has to be modified in advance, as we will discuss in the following sections.

III. SIDE-CHANNEL SOFTWARE WATERMARK

The main idea behind the side-channel software watermark that we first introduced in [3] is to use the power consumption as a hidden communication channel to transmit a watermark. This idea is related to the side-channel-based hardware watermark proposed in [4]. The watermark is hidden in the power consumption of the tested system and can only be revealed by a verifier who possesses the watermark secret. The watermark consists of two components

- combination function
- leakage generator

and is realized by adding instructions at the assembly level to the targeted code. The combination function uses some known internal state of the program and a watermark constant to compute a one-bit output. This output bit is then leaked out (transmitted) via the power consumption using a leakage generator. The leakage generator is realized by one or several instructions whose power dissipation depends on the value of the combination function. This results in a power consumption that depends on the known internal state, the combination function and the watermark constant. To detect the watermark, the verifier can use his knowledge of the watermark design to perform a correlation power analysis (CPA), similar to a classic side-channel attack [13]. If this power analysis is successful, i.e., the watermark signal is detected in the power traces, the verifier can be sure that his watermark is embedded in the device. In many applications this will imply that a copy of the original software is present.

In the following, we will first describe the details of our proof-of-concept implementation before we explain how this watermark can be detected by means of a side-channel analysis.

A. Implementation

To show the feasibility of our approach we implemented a side-channel watermark on the ATmega8 microcontroller. As mentioned, the watermark consists of a combination function and a leakage generator. In our case the input to the combination function is a 16-bit internal state and a 16-bit watermark constant. The combination function computes a one-bit output which is leaked out using a leakage generator. There are many ways to implement a combination function and the combination function introduced in this paper should only be seen as an example. In our implementation we chose a very small and compact combination function that only consists of four assembler instructions. The 16-bit input to the combination function is separated into two byte values \(in_1\) and \(in_2\). In the first step, \(in_1\) and \(in_2\) are each subtracted from the one-byte watermark constants, \(c_1\) and \(c_2\). In the next step the two one-byte results of these subtractions are multiplied with each other. The resulting two byte value \(r\) from this multiplication is again separated into two one-byte values \(r_0\) and \(r_1\). \(r_0\) and \(r_1\) are then multiplied with each other again.

\[
\begin{align*}
    r &= (in_1 - c_1) \cdot (in_2 - c_2) = r_0 | r_1 \\
    res &= r_0 \cdot r_1
\end{align*}
\]

The output of the combination function is the 8th least significant bit of the result \(res\) of this multiplication. The corresponding assembler code can be found below.

```
subi in1,35 ; subtracts constant from in1
subi in2,202 ; subtracts constant from in2
mul in1,in2 ; multiplies in1 and in2 and
            ; stores the result in R0 and R1
mul R0,R1 ; multiplies R0 and R1
```

The registers \(in_1\) and \(in_2\) are used as the internal states and the integers 35 and 202 are the two watermark constants \(c_1\) and \(c_2\). The instruction \texttt{subi in1,35} subtracts the constant 35 from \(in_1\) and stores the result back in \(in_1\). In the ATmega8 instruction set the two-byte result of the multiply instruction \texttt{mul} is always stored in \(R0\) and \(R1\). The result of the combination function is the most significant bit in \(R0\).

A leakage generator is used to leak out the output bit of the combination function. In our implementation we used a conditional jump as our leakage generator. If the output bit is 0, we compute the two’s complement of the register \(R0\), otherwise no operation is executed. We furthermore store the result of \(R0\) in the memory. Below is the corresponding assembler code:

```
SBRC R0,7 ; skip next instruction if
           ; bit 7 in register R0 is 0
neg R0 ; compute 2’s complement of R0
st Z,R0 ; store R0 in the RAM
```

Recall that the output of the combination function is the most significant bit of \(R0\). \texttt{SBRC R0,7} checks the most significant bit of \(R0\) and skips the next instruction if this bit is 0. Otherwise the \texttt{neg} instruction is executed, which computes the two’s complement of \(R0\). In the last step, the value of \(R0\) is stored in the memory.

The leakage generator helps the verifier to detect the watermark. The difference in the power consumption between the case that the \texttt{neg} instruction is executed in comparison to the case the instruction is skipped is very large. This makes detecting the watermark using side-channel analysis straightforward.

We were also able to successfully detect the watermark without any leakage generator. This is due to the fact that the power consumption of the last multiply instruction is higher if the output bit is 1 compared to 0. However, the leakage generator makes detection much easier and can also protect against reverse-engineering and code-transformation attacks (see Sect. V). In the next sub-
section we will give more details on how the watermark detection works.

B. Watermark Verification

To detect the watermark we use a correlation power analysis (CPA). The main idea of a correlation power analysis is to exploit the fact that the power consumption of a device depends on the executed algorithm as well as on the processed data. However, this data dependent power consumption might be too small to be observed with a simple power analysis as it has been used in Section II. Therefore, a lot of traces are measured in a CPA and then statistical methods are used to extract the wanted information. In a classical CPA setting the goal is to retrieve a secret key. In the watermark case, the verifier does not want to retrieve a secret key but wants to verify whether or not his watermark is present. To do this, the verifier first collects power traces with different inputs of the system under test. For each trace the verifier computes the known internal state that is used as the input to the combination function (in our implementation this was the value of register in1 and in2). The internal state used for the watermark should be a varying state that is predictable for the verifier, e.g., a state depending on the input values. The verifier uses this internal state and the watermark constants to compute the output of the used combination function for each input value and stores these values as the correct hypothesis. He repeats this procedure \( n \) times by using different watermark constants or combination functions. At the end the verifier has \( n \) hypotheses with \( n \) different watermark constants or combination functions, where one of the hypotheses contains the correct watermark constant and combination function.

In the last step the verifier correlates the hypotheses with the measured power traces. If the watermark is embedded in the tested device, a correlation peak should be visible for the hypothesis with the correct watermark constant. This is due to the fact that the correct hypothesis is the best prediction of the power consumption during the execution of the leakage circuit. The reference [5] gives a more detailed introduction to this side-channel analysis method. The result of the CPA on our example implementation can be found in Figs. 7 and 8. In Fig. 8 we can see that detecting the watermark is possible with less than 100 measurements.

Other microcontrollers will have a different power behavior and therefore the number of traces needed to detect the watermark might vary from CPU to CPU. It should be noted that a few hundred traces are easily obtained from most practical embedded systems, and it is reasonable to assume that a verifier can use much more measurements if needed. Hence, even if the signal-to-noise ratio might decrease for other microcontrollers, it is safe to assume that detection of this kind of watermark will in most cases be possible. It should also be noted that the length of the code that is being watermarked does not have an impact on the signal-to-noise ratio of the detection. The number of instructions that are executed before or after the watermark does not make any difference in this type of side-channel analysis.

C. Triggering

One important aspect of the software watermark detection is the triggering and alignment of the power measurements. To take a power measurement we need to have a trigger signal that will indicate the oscilloscope to start the measurement. In practice, a communication signal or the power-up signal is usually used as the trigger signal for the oscilloscope. Other possible trigger points might be an unusual low or high power consumption, e.g., when data is written to a non-volatile memory position, the microcontroller wakes up from a sleep mode or a coprocessor is activated. Modern oscilloscopes have advanced triggering mechanisms where several trigger conditions can be used at once, e.g., a specific signal from the bus followed by an unusual high power consumption, etc. Because these trigger points might not be close to the inserted watermark, the verifier might need to locate and align the watermark in a large power trace. With some knowledge of the underlying design, it is usually possible to guess a time window in which the watermarked code is executed. Looking for power patterns that, e.g., are caused by a large number of memory lookups or the activation of a coprocessor can also help to identify the time window where the watermark is executed. Once this time window is located, alignment mechanisms such as simple pattern matching algorithms [13] or more advanced methods such
as [16], [20] are used to align each power trace with each other.

Basically, the problem of triggering and alignment of the power trace for a side-channel watermark is similar to the problem of triggering and alignment of the power traces in a real-world side-channel attack. Often in a real-world side-channel attack the attacker has actually less knowledge of the attacked system than the verifier has for detecting the watermark. The verifier knows the code and flow of his watermarked program while in a real-world attack the attacker can often only guess how it is implemented. We would therefore like to refer to the area of real-world side-channel attacks for more details on the feasibility of triggering a measurement in practice [9], [14], [15], [17]. The alignment of power traces will be addressed in more detail in Section V-B when we describe how to overcome the insertion of random delays as one of the possible attacks on our watermark.

IV. PROOF-OF-OWNERSHIP

The watermark discussed in the previous section only transmits one bit of information: either the watermark is present or not. This is very helpful to detect whether or not your code was used in an embedded system. However, it is not possible for the verifier to prove towards a third party that he is the legitimate owner of the watermark. This is due to the fact that the watermark itself does not contain any information about the party who inserted the watermark. Therefore, anyone could claim to be the owner of the watermark once he detects a watermark in a system. We call the goal to be able to prove towards a third party that you are the legitimate owner of the watermark proof-of-ownership. In this section, we will show how we can expand the watermark idea from Sect. III to also provide proof-of-ownership.

The idea to establish proof-of-ownership is to modify the side-channel watermark in a way that the watermark transmits a digital signature that can uniquely identify the owner of the watermark. One nice property of the side-channel watermark is that the watermark is hidden in the noise of the power consumption of the system. Without the knowledge of the owner the watermark can still not be detected. So we have already established a hidden communication channel. In Fig. 7(a) we can observe a positive correlation peak while the leakage circuit is being executed. Recall that the leakage circuit is designed in such a way that the power consumption is higher when the output of the combination function is ‘1’. If we change the leakage generator in a way that the power consumption is higher when the output bit is ‘0’ instead of ‘1’, then the correlation will be inverted. That means that we would not see a positive, but a negative correlation peak. We can use this property to transmit information. If the bit we want to transmit is ‘0’, we invert the output bit of the combination function. If it is ‘1’, we do not invert the output bit. By doing so, we know that ‘1’ is being transmitted when we see a positive correlation peak and ‘0’ when we see a negative correlation peak. We can use this method to transmit data one bit at a time. We tested this kind of watermark by using the same combination function as discussed in Sect. III but exchanged the leakage generator. We stored an 80-bit signature that we want to transmit with our watermark in the program memory. We then load the signature, one byte at a time, from the program memory and subsequently add the output bit of the combination function to the bit we want to transmit. The resulting bit is then leaked out using a conditional jump, just as it has been done in Sec. III-A.

The same detection method as explained in Sect. III-B is used to detect the watermark and to read out the transmitted signature. Figure 9 shows the result of this correlation based power analysis. The positive and negative correlation peaks over time represent the transmitted signature. The resulting watermark is still quite small. In our example implementation the watermark consists of only 15 assembler instructions for the leakage generator and only 4 instructions for the combination function. We also used 80 bits of the program memory to store the digital signature. If storing the signature in the program memory is too suspicious, it is also possible to implement the leakage generator without a load instruction by using constants. This might increase the code-size a bit, but it is still possible to program the leakage generator with around 30 instructions for a 80 bit signature on an 8-bit microcontroller. In a 16-bit or 32-bit architecture, smaller code sizes can be achieved.

![Side-channel watermark that transmits an ID. Positive correlation peaks indicate that a ‘1’ is being transmitted, negative correlation peaks indicate a ‘0’. In this Figure we can see how the hexadecimal string “E926CFFD” is being transmitted.](image)

Fig. 9. Side-channel watermark that transmits an ID. Positive correlation peaks indicate that a ‘1’ is being transmitted, negative correlation peaks indicate a ‘0’. In this Figure we can see how the hexadecimal string “E926CFFD” is being transmitted.

V. ROBUSTNESS AND SECURITY ANALYSIS OF THE SOFTWARE WATERMARK

In the previous section we have introduced our side-channel watermarks and showed that we are able to reliably detect the watermarks. However, so far we have not talked about the security of the watermark. Traditionally, the security of watermarks towards different attacks is called robustness. To the best of our knowledge there does not exist a completely robust software watermark that can withstand all known attacks. For software watermarks,
and especially side-channel watermarks, it is very difficult to quantify the robustness of the watermark. We do not claim that our watermark is “completely robust” or secure — given sufficient effort the side-channel software watermark can be removed. In the following, we will introduce our security model and describe some possible attacks against the system. We will provide arguments why these attacks can be non-trivial in practice. Hence, we will show that the watermark — although not impossible to remove — still represents a significant obstacle for attackers.

In the security model of the software watermark three parties are involved: The owner of the watermark who inserted the watermark, the verifier who locates the watermark in a suspected device and an attacker who tries to remove the watermark from a software code. The attacker has only access to the assembler code of the watermarked program. The attacker does not know the design of the combination function as well as what part of the assembler code implements this combination function and which internal states or constants are being used in this combination function. This knowledge is considered the watermark secret. The verifier needs to be a trusted third party who shares the watermark secret with the owner of the watermark. A successful attack is defined as follows:

A transformation of the watermarked software code that (1) will make it impossible for the verifier to locate the watermark with means of side-channel analysis and (2) does not change the functionality of the software program.

Hence, an attacker was unsuccessful if either the verifier is still able to detect the software watermark or the resulting software code does not fulfill the intended purpose of the program any longer. We will discuss three different attack approaches to remove the watermark from the assembler code:

Reverse-engineering attack: In a reverse-engineering attack the attacker tries to locate the assembler instructions that implement the watermark using reverse-engineering techniques so that he can remove or alter these instructions.

Code-transformation attacks: In a code-transformation attack, the attacker uses automated code-transformations to change the original assembler code in a way that the resulting code is still functioning correct but the watermark detection is impossible.

Side-channel attacks: In a side-channel attack the attacker tries to use side-channel techniques to locate the side-channel signal in the power consumption. This gives the attacker the knowledge of the location of some of the watermark instructions (e.g., the leakage generator).

In the following we discuss each of the three attacks in more detail.

A. Reverse-Engineering Attack

If the attacker can reverse-engineer the entire code and identify the purpose of each instruction and function, the attacker also knows which instructions are not directly needed by the program and which are therefore possible watermark instructions. However, complete reverse-engineering of the assembler code can be very difficult and time consuming, especially in larger programs. Furthermore, complete reverse-engineering might be more expensive than actually implementing it, making product piracy not cost effective if reverse-engineering is needed. An attacker can try to locate the watermark without reverse-engineering the entire code. For example, the attacker could use techniques such as data-flow diagrams to detect suspicious code segments which he can then investigate further. The complexity of such attacks depends on the attackers reverse-engineer skills as well as on the way the watermark is embedded in the code.

We believe that due to the small size of the watermarks, locating the watermarks with methods of reverse-engineering can be very expensive for the attacker. Especially in larger designs, which are usually more attractive for software theft, this can be very difficult. Another attractive property of the side-channel watermarks is that they are hidden in the power consumption of the system. This means that an attacker cannot tell whether or not a watermark is present in the code. So even if she locates and removes one or several side-channel watermarks from a code, she cannot be sure if there are not still more watermarks present in the code. Considering the small size of only 5-10 assembly instructions for some watermarks, adding multiple watermarks is still very economical. This may discourage attackers from stealing the code as reverse-engineering the entire code is necessary to ensure that all watermarks have been removed.

B. Code-Transformation Attacks

In an automated code-transformation attack, a software is used to change the program code without changing the semantically correct execution of the program. Examples for code-transformations are recompiling the code, reordering of instructions and obfuscation techniques such as replacing one instruction with one or more instructions that have the same result. Code-transformations can be a very powerful attack tool for disabling software watermarks as has been shown in [11], where all tested static software watermarks for Java bytecodes have been successfully removed with standard obfuscation tools.

Let us first consider the impact of reordering of instructions and the insertion of dummy instructions on our side-channel watermark. If these methods are used by the attacker, they can have the effect that the leakage generator is executed in a different clock cycle compared to the original code. For the detection this means that the correlation peak will be at a different clock cycle. However, the correlation peak will be as visible as without the reordering as inserting a static delay does not decrease the signal-to-noise ratio. Therefore, simple reordering and the insertion of dummy instructions cannot prevent the verifier from detecting the watermark.

However, if the attacker does not add a static but a random delay this will have a negative impact on
the watermark detection. Random delays have the effect that the measurement traces are not aligned with each other, i.e., the clock cycle where the leakage generator is executed varies from measurement to measurement. Unaligned traces hamper side-channel analysis but the detection can still be successful if enough traces are aligned with each other [13]. It is not always easy to insert efficient random delays into the code, e.g., a source of randomness is needed and simply measuring the execution time of a program might give indication of the random delay introduced. Furthermore, the verifier can use alignment methods to detect such misalignment and remove the random delays. By using these alignment methods the verifier has a good chance to counteract the random delays, especially if the delays are inserted several clock cycles before the leakage generator. Due to the fact that the attacker does not know the location of the leakage generator this is very likely. Otherwise the attacker would need to insert a lot of random delays which will hurt the performance.

To show the power of alignment techniques to counteract random delays we changed our experiment by inserting random delays. In our first approach we added our own random delays by using a timer interrupt that would pseudo-randomly trigger every 1-128 clock cycles. We used a S-Box to generate our pseudo-random numbers and an externally generated 8-bit random number as its initialization for each measurement. These random interrupts did not provide much of an obstacle and with a simple pattern matching algorithm [13] we could detect the watermark.

The result of the CPA is shown in Fig. 10(a). To make our experiment more credible we also implemented the random delay based side-channel countermeasure presented at [8] to our watermarked AES. This countermeasure, called improved floating mean, inserts random delays at fixed positions but with a varying length. The initial state of the used PRNG is an externally generated 64-bit number. We again used a pattern-matching alignment algorithm and peak extraction before performing our CPA. The result of this analysis is shown in Fig. 10(b). The correlation coefficient decreased for this analysis compared to the original watermarked AES implementation from around 0.85 to 0.5 but this correlation value is still very large. These results aim to show that it is not simple to insert random delays to defeat the watermark detection. With more improved alignment methods (e.g. [16], [20]) better results could probably be achieved. Furthermore, in [18] it was demonstrated that the random delays can be removed without a big decrease in the correlation coefficient with methods similar to the ones described in Section II.

By replacing instructions, an attacker might change the power profile of a code. For example, instead of using the
decrement instruction DEC to decrease a register value, the subtract with constant instruction SUBI could be used. These instructions have a different power profile. However, even if the attacker can change the power profile of the code significantly, this does not impact the CPA. The power consumption of the clock cycles before or after the watermark do not have impact on the CPA correlation. As long as there is a difference in power consumption according to the output of the combination function this difference can be used to detect the watermark using a CPA. For example, just the transmission of the output of the combination function over the internal bus usually leaks enough data-dependent power consumption to be used in a side-channel analysis, regardless of the actual instruction that is being executed.

A code-transformation that removes the output of the combination function on the other hand would be successful. But every code-transformation algorithm needs to make sure that the resulting code does not change the semantically correct execution of the program. Therefore, it needs to be ensured that for the compiler or code-transformation algorithm the watermark value is considered needed. This can be done by storing the output or using it in some other way. In this case any code-transformation attacks will be unsuccessful as removing the watermark value would destroy the semantical correct execution of the program from the view of a compiler.

Adding additional side-channel watermarks to a program can be seen as a code-transformation as well. A side-channel watermark should not change the state of the program that is being watermarked to ensure that the watermark does not cause software failures. Hence, additional watermarks will not change the combination function of a previously inserted watermark. They might only introduce some either static or data-dependent delays. For this reason, it is possible to add multiple watermarks into a design without interference problems.

---

3We used the parameters provided in [8] for our implementation of improved floating mean with three dummy rounds before the encryption and inserted the watermark in the main AES encryption function. 29 random delays, each varying between 24-536 clock cycles in steps of two, are executed before the first execution of the watermark.

4This assumes that the power consumption of the internal bus is correlated with the hamming weight of the transmitted bits, which is usually the case for microcontrollers[13].
C. Side-Channel Attacks

If the attacker can successfully detect the watermark using a side-channel analysis, the attacker also gains the knowledge of the exact clock cycles where watermark instructions (e.g., the leakage generators) are executed. In this case the attacker only needs to remove or alter these instructions to make the watermark detection impossible. Therefore, the watermark should only be detectable by the legitimate verifier who possesses the watermark secret.

The attacker can try to discover the watermark secret by performing a brute-force side-channel analysis in which he tries every possible watermark secret. But the attacker faces two problems with this approach: the big search space of possible watermarks and false-positives. The size of the search space of possible watermark secrets depends strongly on the application, the size of the watermark, and the architecture of the microcontroller. The application that is being watermarked determines how many internal states can be used as inputs to the combination function and the size of the watermark determines how many operations the combination function performs. Finally, the number of available instructions and functions that can be used for the watermark also influences the search space.

In the following, we give a rough estimation of a possible search space for our example application of an AES-128 encryption on the 8-bit ATmega8 microcontroller. The AES encryption program has two 16-byte inputs, the plaintext and the key, and one 16-byte output, the ciphertext. Let us assume that the designer of the watermark can use the 16-byte input for his internal state of the watermark. For simplicity we assume that the combination function consists of 10 basic instructions using the internal states and two 8-bit watermark constants as inputs. Furthermore we assume that only the six ATmega8 instructions addition, subtraction, AND, OR, exclusive-OR, and multiplication can be used. Using these parameters the lower bound of possible different combination functions is roughly $2^{75}$.

Besides the large search space an attacker has also to face the problem of false-positives. If an attacker tries $2^{75}$ different watermark secrets it is likely that the attacker will see some correlation peaks that are not due to the actual watermark. One reason for a correlation peak might simply be noise as statistically some hypotheses will generate greater correlation peaks than others. Such peaks are usually called ghost peaks in the literature. These correlation peaks should be smaller than the actual correlation peak due to the watermark if enough traces are used. However, if the attacker has not discovered the watermark yet he does not know how high the correlation peak of the watermark is supposed to be and might therefore falsely suspect wrong parts of the design to be the watermark. The second reason why a false-positive might appear is that some part of the actual program that is being watermarked might be linearly related to a possible watermark. In a brute-force approach all possible operations on the internal states are tested. Therefore, it is more than likely that one or several of the tested combination functions are identical or linearly related to parts of the actual program. In this case, correlation peaks appear that will indicate a possible watermark at a location where there is no watermark embedded.

To summarize, detecting the watermark using side-channel analysis can be quite complicated for the attacker. Using small (and possible multiple) watermarks will increase the problem of false-positives while the search space becomes too big in practice if larger watermark secrets (more operations and/or more possible internal states) are used. In our opinion, it seems that using a reverse-engineering or code-transformation approach is more promising to remove the watermark in practice.

VI. Conclusion

In this work we introduced three very efficient and cost-effective ways to detect software plagiarism and piracy in embedded systems. The biggest advantage of these methods are that they do not require access to the suspicious program code. This property is very useful for embedded systems as the access to the program code is usually restricted by program memory protection mechanisms. Hence, our methods enable a verifier to efficiently test many embedded systems towards software plagiarism by simply measuring the power consumption of the devices under test. In our first approach, we achieve this by deriving the Hamming weight of the running opcode from the taken power measurements. This can be done without the need to modify the program code, i.e., no watermark needs to be inserted. The Hamming weight method can detected one-to-one copies of software code very easily. We also showed that this detection mechanism can still be successful if the attacker makes some changes to the program code. For this case we showed how string matching algorithms can be applied to the Hamming weight method to determine how much the tested code is similar to the reference code. If the code is very similar to the reference code, this is a very good indicator for software plagiarism.

For an increased robustness and easier detection we have introduced the side-channel software watermarks. These watermarks can be inserted at the assembly level and introduce only a very small overhead in terms of code size and runtime. Like the Hamming weight method, these watermarks can be detected by simply measuring the power consumption of the device under test. Practical experiments showed that only as few as hundred traces were needed to clearly detect the watermark. We have also discussed why these watermarks are very robust to code-transformations and other attacks. Furthermore, the side-channel software watermark can be extended to transmit a digital signature. Such a signature can be used in court to proof the legitimate ownership of the watermark. Hence, this watermark can not only detect the software plagiarism, but can also be used to provide proof-of-ownership.

References