Side-Channel Watermarks for Embedded Software

Georg T. Becker *, Wayne Burleson*, Christof Paar*†,
* University of Massachusetts Amherst, USA
†Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany
Email: becker@ecs.umass.edu, burleson@ecs.umass.edu, Christof.Paar@rub.de

***Abstract*—In this paper we introduce a new software watermarking mechanism for the embedded environment. The newly proposed software watermarking mechanism can be added at the assembly level and hides the watermark in the power consumption of the device. By using side-channel analysis techniques, the verifier can reliably detect his watermark in the power traces of the device. This new approach is especially well suited for embedded microcontrollers that have program memory protection. In comparison to other software watermark mechanisms a verifier does not need to have access to the software code or data memory to detect the watermark. This makes the detection of the watermarks very efficient for embedded applications in which access to the program code or data memory is very restricted. Our watermark method can therefore serve as a very easy and cost efficient way to detect software theft for embedded applications.**

I. INTRODUCTION

Product piracy and Intellectual Property (IP) theft are problems that have become a major concern for many industries. In this paper we focus on the illegal copying, selling or using of software for embedded systems. To protect against illegal copying, different techniques have been proposed. These techniques can be divided into two groups: Techniques that try to prevent the copying process and techniques that can detect and proof the abuse afterwards. Examples for techniques that try to prevent illegal copying are program memory protection mechanisms which are very common in microcontrollers as well as program memory encryption [6]. Software watermarks on the other hand do not prevent attackers from copying the program but are used to detect a fraud. In this work we present a software watermarking mechanism that can be inserted at the assembly level and that is especially useful for embedded applications. Most software watermarking mechanisms need either the software code to detect the watermark [4] or access to the memory structure during execution [7][2]. However, in the embedded environment it is not easy to access the suspected code because in many cases, especially in those systems where code copying is a relevant threat, the memory is protected from unauthorized read operations. Another drawback of many previously proposed methods is that they have proven to be not very robust against code-transformation attacks[4]. In [4] every tested static software watermark mechanism was successfully removed using standard code-transformation software. Our side-channel software watermarks address these problems by providing a software watermarking mechanism that can be detected without access to the program code and that is very robust to code-transformations. This is achieved by hiding the watermark in the power consumption of the device. A verifier who wants to test if an embedded system is running their software can verify the watermark by simply measuring the power consumption. No access to the program memory is needed. The proposed software watermarks are closely related to the hardware watermarks introduced in [1]. For a brief overview of other software watermarking mechanisms we refer the reader to [8] and [3].

This work was supported by the NSF Grants 0916854, 0964641, and 0923313.

The remainder of this paper is structured as follows: First we define the goals of software watermarks before we introduce our new side-channel based software watermarking mechanism. We then present the experimental results of our proof-of-concept implementation. In Section IV we discuss the security and robustness of the watermarking technique before we conclude the paper with a short summary of the achieved results.

II. SOFTWARE WATERMARK MODEL

To avoid any misunderstanding we will first introduce the goal of our software watermarking scheme. The objective of software watermarking is to detect and proof the illegal use of your software code. The party who tries to detect his watermark in a design is called verifier. We denote the person who tries to illegally use the software code as the attacker. The goal of the verifier is to distinguish whether or not his watermark is embedded in a design. The goal of the attacker is to prevent the verifier from detecting the watermark. Software watermarks can have different requirements for the verifier. In many previously proposed watermarking techniques the verifier needs to have access to the software code or access to the memory during the execution of the software to detect the watermark. In our software watermark model, the verifier only needs to have physical access to the device that runs the suspected software. Hence, the verifier does not need to have direct access to the code or the memory.

The attacker can try different methods to make the detection of the watermark impossible, e.g. automated code-transformations such as recompiling the assembly code, or reverse-engineering to locate the watermark instructions. We discuss some possible attack scenarios on our software watermark in Section IV. In the following we now describe the details of our side-channel based software watermark.

III. Watermark design

The main idea behind the software watermark introduced in this work 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 introduced in [1]. 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 is realized by adding instructions at the assembly level to the targeted code. The watermark consists of three components, a watermark key, a combination function and a leakage generator. The combination function uses some known internal state of the program and the watermark key to compute a one bit output. This output bit is then leaked out (transmitted) over 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 key. Figure 1 gives an illustration of the watermark design. To detect the watermark, the verifier can use her knowledge of the watermark construction and key to perform a differential power analysis (DPA), similar to a classic side-channel attack [5]. 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 first provide details of the used combination function and leakage generator of our first proof-of-concept implementation. After that we explain how this watermark can be detected by the means of a side-channel analysis.

A. Implementation

The watermark is implemented on a 8-bit PIC16F687 in assembly language. In our proof-of-concept implementation we used an implementation of the Keeloq encryption algorithm as our original software that we protect with a watermark. The Keeloq implementation takes a 32 bit plaintext as input and returns a 32 bit ciphertext as the output (for the watermark only the input is relevant). We used the 32 bit plaintext from the Keeloq implementation as the known internal state that is used as an input to our combination function in addition to a fixed 64 bit watermark key. We denote the four bytes of the plaintext as $p1, ..., p4$ and the 8 bytes of the watermark key as $k1, ..., k8$. The combination function uses $p1, ..., p4$ and $k1, ..., k8$ to compute a one byte value h as follows:

$h = ((k1 \oplus p1) \& (k2 \oplus p2)) \oplus ((k3 \oplus p3) \& (k4 \oplus p4)) \oplus ((k5 \oplus p1) \& (k6 \oplus p3)) \oplus ((k7 \oplus p2) \& (k8 \oplus p4))$

From this one-byte value h the Hamming weight is computed. The output of the combination function is the least significant bit of the Hamming weight of h.

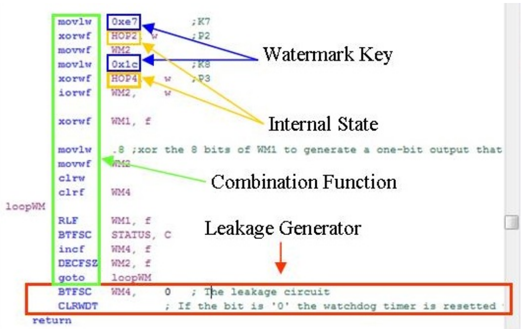


Fig. 1. Illustration of the different components of the software watermark (only a part of the watermark code is shown here). A watermark key, some known input and a combination function are used to compute a one-bit value. This one-bit value is leaked out using the leakage-generator which in this case is a conditional watchdog timer reset.

There are many ways to implement a combination function and this specific combination function should only be seen as an example. Due to the restrictions of the PIC16F687 instruction set and the 64 bit watermark key, our example implementation consisted of 40 lines of code. However, much smaller watermarks can be achieved, e.g., on an ATmega8 microcontroller we used the multiply instruction to make a much more lightweight combination function that consisted of only 4 instructions. In Section IV we will discuss different attacks on the watermark system and will explain more about the choice of a combination function. We will also show why a much smaller watermark key than 64 bits can be used.

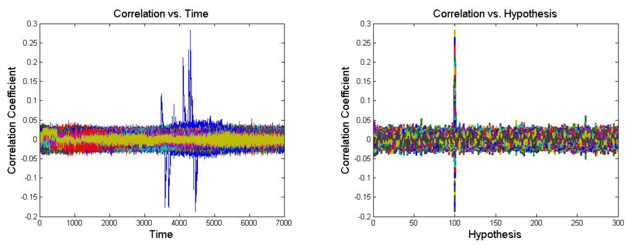
The one-bit output from the combination function is leaked out over the power consumption using a leakage generator. We used a conditional jump with a watchdog timer reset as our leakage generator. When the least significant bit of h is '0' the watchdog timer is reset, otherwise a *nop* instruction is executed. Because a watchdog timer reset consumes more power than a *nop* instruction, the power consumption for a '0' is higher than for a '1' during the execution of the leakage generator. This conditional power consumption is used to detect the watermark. We note that it would also be possible to use the combination function without any leakage generator. This is due to the fact that the Hamming weight of the output of the combination functions generates enough leakage to be detected by a differential power analysis. However, the leakage generator can be very useful to prevent code-transformation attacks against the watermark (see Section IV).

B. Detection

To detect the watermark a differential power analysis (DPA) is preformed just as it is known from side-channel attacks [5]. A DPA is based on the idea that the power consumption of a device depends on the executed algorithm as well as on the processed data. To detect the watermark, the verifier first records many power traces for different input values. For each of the power traces the verifier computes the output of the combination function with the given inputs and the watermark key and stores this value as the correct hypothesis. The verifier then computes k additional hypotheses by using different watermark keys for computing the watermark output. In the last step the verifier correlates all hypotheses with the power traces. If the watermark is embedded in the tested device then a correlation peak should be visible for the hypothesis with the correct watermark key compared to the false hypotheses. This is due to the fact that the hypothesis with the correct key is the best prediction of the power consumption of the watermark.

To proof the feasibility of our method we have implemented the above watermark on a PIC16F687 microcontroller and measured the power traces for 10.000 different input values. We have generated 300 different hypothesis using different watermarking keys. The results of the power analysis can be seen in Figure 2(a) and Figure 2(b). The hypothesis with the correct watermark key generated a significant correlation peak while there was no peak for false watermark keys. In Figure 2(a) clear correlation peaks for the correct hypothesis at different times can be observed. Several assembler instructions depend on the output value of the combination function, not only the watchdog reset that is used as our leakage generator. This explains the different correlation peaks over time for the correct hypothesis. Figure 3 shows the correlation for different numbers of measurements. In this Figure we can see that only a few hundred measurements are enough to detect the watermark. This shows the feasibility of our watermark detection approach.

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. How many instruction are executed before or after the watermark does not make any difference in this type of side-channel analysis. Therefore it is also possible to insert many different side-channel watermarks into one code without any interference problems.



(a) (b)

Fig. 2. The result of the side-channel analysis with (a) respect to time and (b) respect to different watermark-key hypothesis where hypothesis number 100 is the correct one.

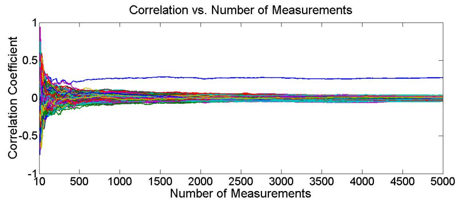


Fig. 3. The results of the side-channel analysis with respect to number of measurements. Even with a few hundred measurements the correct hypothesis can be clearly detected.

IV. Robustness and security analysis

In Section III we have shown that it is possible to reliably detect side-channel software watermarks. But so far we have not discussed the robustness of the watermark and its resistance against possible attack scenarios. A successful attack against the watermark would mean to make the detection of the watermark impossible for the verifier. We will discuss the robustness of our proposed watermarks against three different types of attacks: reverse-engineering attacks, code-transformation attacks and side-channel attacks.

A. Reverse-engineering attack

In a reverse-engineering attack the attacker tries to identify instructions that belong to the watermark and that do not have impact on the correct execution of the software. If the attacker can identify such watermark instructions he can simply remove them. Removing these instructions will destroy the watermark and make the detection of the watermark impossible. However, this attack can be quite difficult in larger programs. One can argue that large code bases, e.g., those used in network routers, tend to constitute a higher monetary value, so that protecting them is particularly relevant in practice. The watermark consists of only a few instructions and therefore the attacker might need to reverse-engineer large parts of the software to locate the watermark. But reverse-engineering at the assembly level is a very difficult and time consuming task. Therefore, in many cases complete reverse-engineering is not economical because the costs for reverse-engineering might be too high. How difficult the reverse-engineering attack is, highly depends on the software that is been watermarked as well as on the watermark size. Especially watermarks consisting of short code sequences, as introduced earlier, can be very difficult to detect.

B. Code-transformation attacks

The software watermark should be resistant against automated code-transformations such as reordering of instructions, recompiling and instruction substitutions that do not change the semantically correct execution of the program. Let us first consider the reordering of instructions and insertions of dummy instructions. Simple reordering of the instructions or the insertion of static dummy instructions has the result that the watermark is computed and leaked out in a different clock cycle than in the original version. This will change the result of the DPA in the way that the correlation peak will now occur at a different time. However, the signal-to-noise ratio of the watermark will not change and the correlation peak will be as visible as without the reordering.

If an attacker inserts random and not fix delays, the watermark will be computed at a different clock-cycle for every trace. Randomized execution is a known countermeasure against side-channel attacks on cryptographic algorithms and can make the detection much more difficult. Basically, if the traces are not synchronized properly, this acts as noise on the side-channel analysis. As long as enough traces are aligned the detection will still be successful even if other traces are not aligned. So the question whether or not a verifier can detect the watermark depends on his ability to align the watermark traces. The verifier usually uses recognition patterns to align the traces. By using recognition pattern the verifier has a high chance to detect the delays and to align the traces. This strongly depends on the size of the watermarked code and the place and length of the random delays inserted by the attacker. Furthermore, it is not easy to insert efficient random delays into the code, e.g. a source of randomness is needed and simply measuring the response time will give indications of how long the random delay is.

Other code-transformations such as recompiling will not be effective against the power watermark as long as the correct watermark output is computed. Every code-transformation algorithm needs to make sure that the resulting code does not change the semantically correct execution of the program. Therefore it only 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 such as in our implementation as a conditional watchdog timer reset. In this case any code-transformations attacks will be unsuccessful as removing the watermark value would destroy the semantical correct execution of the program from the view of a compiler.

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 some watermark instructions (e.g. the leakage generator) 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.

If an attacker knows the used combination function and especially which internal states are used for the watermark the attacker can try to reveal parts of the watermark using different side-channel analysis methods. With such methods it might be possible for the attacker to reveal the watermark key and detect which instructions are used for the leakage generator. Hence, we need to assume that the combination function and/or the used internal states are not known by the attacker and are part of the watermark secret. Given the huge space of possible combination functions together with the huge space of possible internal states, even a very small watermark key of 8 or 16 bit can give enough entropy to the watermark secret to make a brute-force approach to detect the watermark infeasible.

V. Conclusion

In this paper we introduced a novel software watermarking mechanism for embedded applications. The watermark is very lightweight in terms of space and runtime overhead. One of the main advantages of the proposed watermark is its ability to detect the watermark without access to the code. This makes it very useful for the embedded environment as in most microcontrollers protection methods are used to prevent unauthorized access to the code. Previously proposed watermarking mechanisms would need expensive and time-consuming reverse-engineering techniques to detect the watermark in a protected embedded application. Measuring the power consumption on the other hand is relatively easy, making these side-channel watermarks very efficient for the use in the embedded environment.

Another advantage of the prosed watermark is its robustness against code-transformations such as recompiling or shuffling of instruction. The biggest vulnerability to this type of watermark are probably reverse-engineering attacks that try to locate the watermark instructions. However, reverse-engineering at the assembly level can be very time consuming and expansive which will discourage attackers. Furthermore, such an attack cannot be executed automatically as it is the case with code-transformation attacks. Side-channel attacks to locate the watermark instructions are only feasible as long as the attacker tries the correct combination function with the correct internal states. But if the combination function and the used internal states are treated as the watermark secret, this should be infeasible due to the huge design space of possible combination functions and internal states.

References

[1] G.T. Becker, M. Kasper, A. Moradi, and C. Paar. Side-channel based watermarks for integrated circuits. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, pages 30 –35, June 2010.

[2] C. S. Collberg, A. Huntwork, E. Carter, G. Townsend, and M. Stepp. More on graph theoretic software watermarks: Implementation, analysis, and attacks. *Inf. Softw. Technol.*, 51:56–67, January 2009.

[3] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *Software Engineering, IEEE Transactions on*, 28(8):735 – 746, August 2002.

[4] J. Hamilton and S. Danicic. An evaluation of static java bytecode watermarking. In *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2010*, pages 1 – 8, San Francisco, USA, October 2010.

[5] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology CRYPTO99, Lecture Notes in Computer Science*, page 388397. Springer-Verlag, June 1990.

[6] D. Marsh. Silicon simplifies embedded-system security. *EDN Europe*, Jul 2008.

[7] W. Zhu and C. Thomborson. Algorithms to watermark software through register allocation. In *Digital Rights Management. Technologies, Issues, Challenges and Systems*, volume 3919 of *Lecture Notes in Computer Science*, pages 180–191. Springer Berlin / Heidelberg, 2006.

[8] W. Zhu, C. Thomborson, and F-Y. Wang. A survey of software watermarking. In *Intelligence and Security Informatics*, volume 3495 of *Lecture Notes in Computer Science*, pages 454–458. Springer Berlin / Heidelberg, 2005.