Mechanism for Software Tamper Resistance: An Application of White-Box Cryptography

Wil Michiels
Philips Research Laboratories
High Tech Campus 34
Eindhoven, The Netherlands
wil.michiels@philips.com

Paul Gorissen
Philips Research Laboratories
High Tech Campus 34
Eindhoven, The Netherlands
paul.gorissen@philips.com

ABSTRACT

In software protection we typically have to deal with the white-box attack model. In this model an attacker is assumed to have full access to the software and full control over its execution. The goal of white-box cryptography is to implement cryptographic algorithms in software such that it is hard for an attacker to extract the key by a white-box attack. Chow et al.[8, 7] present white-box implementations for AES and DES. Based on their ideas, white-box implementations can be derived for other block ciphers as well. In the white-box implementations the key of the underlying block cipher is expanded from several bytes to a collection of lookup tables with a total size in the order of hundreds of kilobytes.

In this paper we present a technique that uses a white-box implementation to make software tamper resistant. The technique interprets the binary of software code as lookup tables, which are next incorporated into the collection of lookup tables of a white-box implementation. This makes the code tamper resistant as the dual interpretation implies that a change in the code results in an unintentional change in the white-box implementation. We also indicate in the paper that it is difficult for an attacker to make modifications to the white-box implementation such that its original operation is restored.

Categories and Subject Descriptors

H.3.2 [**Information Systems**]: Information Storage and Retrieval—*Information Storage*

General Terms

Security

Keywords

Software protection, software tamper resistance, white-box cryptography

DRM'07, October 29, 2007, Alexandria, Virginia, USA.

1. INTRODUCTION

As the need for flexibility and complexity grows, the functionality of devices is more and more implemented in software instead of in hardware. This trend makes it increasingly important to develop techniques for protecting software. The problem with software protection is the severe attack model that we have to deal with. Instead of the conventional 'black-box attack model' in which an attacker has at most access to the input and output of the program, we have to deal with the 'white-box attack model' [8, 7]. This attack model is the strongest conceivable one in which an attacker is assumed to have full access to the software and full control over the execution environment. The reason we have to deal with this strong attack model is that, while the black-box attack model assumes in a communication that the end-points are trusted, a software attack comes generally from the inside; the attacker is typically the user of the software or a virus that has installed itself on the device that is running the software.

Two important problems in software protection are obfuscation and tamper resistance. In the former problem the goal is to protect a software program against reverse engineering by transforming the program into a functionally equivalent one that is harder to understand. In the latter problem, we have to protect against an attacker who tries to make a modification to a software program, such that the software gets a particular, different functionality. As an illustrative example, we mention software that contains a routine for implementing access and permission control. The user of the software may try to disable or change this routine.

Both obfuscation and tamper resistance are very hard problems, and methods with a provable security for practical programs are far beyond the state of the art [2, 17]. This implies that in practice one has to resort to applying multiple complementary tools to tackle the problems. In this paper we present a tool in the toolbox for making software tamper resistant.

The method we propose is an application of 'white-box cryptography' and can be implemented completely in software. White-box cryptography is the discipline that aims at solving the obfuscation problem of how to implement a cryptographic algorithm in software, such that the key cannot be extracted by a white-box attack. This problem is, for instance, relevant for a content provider who broadcasts encrypted content and who wants to prevent a licensed user from illegally putting the decryption key on the Internet. An implementation of a cryptographic algorithm that tries to resist a white-box attack on its key is called a white-box implementation. Chow et al. present white-box implementations for the block ciphers AES and DES [8, 7]. These white-box implementations are based on ideas that can be easily extended to other block ciphers. The white-box implementations of Chow et al. hide the key of the implemented block cipher in a large collection of lookup tables. As

a result, a white-box implementation can be viewed as stand-alone cryptographic algorithm the key of which is given by this collection of lookup tables. This view on a white-box implementation is used when we refer to the key of a white-box implementation. To avoid confusion with the key of the implemented block cipher, the key of a white-box implementation is also called white-box key. The total size of the lookup tables is in the order of hundreds of kilobytes.

The software tamper-resistance technique presented in this paper is an application of white-box cryptography in the sense that the technique makes the correct operation of the white-box implementation of a block cipher dependent on the integrity of software. That is, if an attacker modifies the software, the white-box implementation stops decrypting/encrypting properly. As a result, the tamper-resistance technique requires that the system on which we want to protect software uses a block cipher implementation for which we can derive a white-box implementation. When referring to the security of the technique, we assume that it is the goal of an attacker to modify the protected software without losing the ability to decrypt/encrypt properly. As attack model we assume the white-box attack model.

As a possible application of our technique we mention a DRM client that is implemented in software and that has to validate conditions in a DRM license before it decrypts the corresponding content. The content can, for instance, be encrypted by AES as this block cipher allows a white-box implementation. In a content rental model the DRM license typically prescribes that content may only be decrypted during a specific time window. An attacker may try to get around the license by tampering with the program code that verifies the license. Hence, we want to make this code tamper resistant. It is to be preferred to not only protect the license verification routine, but also (part of) the software that calls this routine. This in order to prevent an attacker from removing calls to the routine.

At a high level, the software tamper-resistance technique works as follows. Let B be the binary of the software that we want to protect. Binary B can be linked and compiled code obtained from higher lever source code, such as C, but it can also be the binary representation of interpreted code, such as byte code in Java. As binary B is just a string of bits, we can also interpret it as a collection of lookup tables. A code fragment of 1024 bytes can, for instance, be interpreted as a lookup table consisting of 256 rows of 4 bytes.

The tamper-resistance technique generalizes the techniques for implementing a block cipher by a white-box implementation. The generalization enables us to add arbitrary tables to a white-box implementation without changing the cryptographic function that it implements. This implies that we can implement the block cipher that runs on the system by a white-box implementation, such that the executable code B, interpreted as a collection of lookup tables, is included into the collection of lookup tables that defines the white-box key. After applying the technique, the code has a dual interpretation: it is both program code and key.

Fig. 1 visualizes how the technique can be applied to the DRM application mentioned above. The decryption algorithm is implemented by a white-box implementation consisting of a collection of lookup tables (the white-box key) and a decryption routine that uses the white-box key to decrypt. In the figure, binary B is the complete software image of the DRM client, i.e., the complete software image is protected by including it in the white-box key. The software image includes the license verification routine and the white-box decryption routine.

If we now make a modification to the program code that we included in the white-box key, then this results in an undesired modification of the cryptographic key: this turns the key into an invalid one. That is, the code is tamper resistant in the sense that its integrity is checked each time that the cryptographic key is used. Moreover, the technique presented includes code into a white-box key in such a way that it is difficult for an attacker to repair the correct operation of a white-box implementation after its key has been invalidated. Compared to existing tamper resistance techniques, this method has the advantage that the protected software program remains in the clear. It does not require that, besides the white-box implementation, parts of it are hidden by code obfuscation techniques.

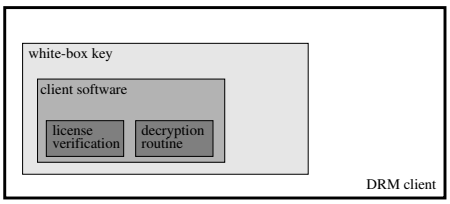


Figure 1: Tamper resistance technique applied to DRM application.

This paper is organized as follows. First, we present in Sect. 2 related work on software tamper resistance. In this paper we use the white-box AES implementation of Chow et al. as a carrier to explain our software tamper resistance technique. However, instead of AES the technique can also be applied to any other block cipher that allows a similar table-based white-box implementation. In Sect. 3 we briefly discuss the white-box AES implementation. In Sect. 4 we introduce our tamper resistance technique, which we named 'Medusa' after the female character from Greek mythology whose glimpse turned any living creature into stone. The security of Medusa is discussed in Sect. 5. We end with some concluding remarks in Sect. 6.

2. RELATED WORK

In this section we discuss related work on software tamper resistance techniques. We restrict ourselves to techniques that, as Medusa, can be fully implemented in software and need no hardware support. The two main approaches for tackling the problem of software tamper resistance are code obfuscation and cryptographic hashing. The goal of code obfuscation techniques is to make it hard for a reverse engineer to understand the program. The idea behind applying obfuscation to achieve tamper resistance is that it is difficult to realize an intended change in functionality if we do not know where and how to change the code.

A possible obfuscation technique is to store the program in encrypted form. The decryption can then be done in either hardware or software. Software solutions are given by Aucsmith [1] and Wang, Kang, and Kim [20]. Their approaches divide the program into cells. At any stage of the execution all but one cell is encrypted. The only cell that is exposed unencrypted is the one the program counter points into. Obfuscation by encryption only protects against a static analysis and not against a dynamic analysis. Furthermore, it implies a severe execution-time penalty.

An alternative approach for obfuscating a program is to apply code transformations that turn the program into one that is functionally equivalent, but more complex and less readable. Collberg, Thomborson, and Low [10] present many of such transformations. Other references are Cohen [9], Wang et al. [19], and Wroblewski [21]. Although to a lesser extent than the encryption approaches, obfuscation by code transformation still incurs an execution-time penalty.

Cryptographic hashing is a second main approach for making code more tamper resistant. Hashing techniques compute hash values of code fragments and include checks in the program that compare these hash values with the predefined values. An incorrect value causes the program to stop working properly. To be effective, hashing has to be combined with obfuscation because if we are able to identify the computations of the hash values or the checks on them, then it is easy to get around the protection mechanism. However, by applying obfuscation, we also get the disadvantages of it, such as an execution-time penalty. Furthermore, because hash-computations and the checks on their outcome have a very typical format, a good obfuscation is problematic.

Tamper resistance techniques that are based on hashing are presented by Horne et al. [14], Chang and Atallah [5], and Chen et al. [6]. Also the approach presented by Wang, Kang, and Kim [20], to which we already referred to, employs hashing.

All techniques discussed in this section incur an execution-time penalty. An approach to reduce this penalty is to construct small secure processes that are used as verification engines for the entire program. The costly techniques then only have to be applied to these small processes. This approach is adopted by Aucsmith [1] and Blietz and Tyagi [4].

We conclude this section with the remark that our approach for making software tamper resistant is related to a paper of Yuval. Yuval [23] also proposes to interpret software as lookup tables within a block cipher. In order to save on storage space, he suggests to define the lookup table that implements an S-box as a code fragment.

3. WHITE-BOX AES IMPLEMENTATION

As mentioned in the introduction, we use the white-box AES implementation of Chow et al. [7] as a carrier to explain Medusa. Our technique is, however, not restricted to AES. The ideas of Chow et al. can be applied to other block ciphers in order to derive table-based white-box implementations to which we can apply Medusa.

We note that attacks have been published for extracting the 128-bit AES key or the 56-bit DES key from the white-box AES and the white-box DES implementations of Chow et al. [3, 13, 15, 16, 22]. However, these implementations can still provide an effective protection of the key; by intertwining the large collection of lookup tables and by applying code obfuscation techniques to the white-box implementation the task of extracting a key can be complicated considerably. In Sect. 2 we provided references to papers describing code obfuscation techniques. Although a formal metric to express the quality of an obfuscation lacks, we believe that a white-box implementation allows a more effective obfuscation than a regular implementation of a block cipher. Intuitively, it is easier to hide the exact value of the individual lookup tables in a white-box key of several hundreds of kilobytes than to hide a standard DES and AES key of only 56 and 128 bits, respectively.

Instead of diving directly into the details of the white-box AES implementation as proposed by Chow et al. [7], we first give a higher level discussion to explain its basic idea. First, each round of AES is implemented by a series of lookup tables. That is, the output of a round is computed by performing table lookups only, where the input to a lookup table is either the input to the round or the output of another lookup table. Such an implementation can be modeled by a network of lookup tables, where an arc from table T to T' means that the output of table T is used as input to table T'.

To arrive at a white-box implementation, Chow et al. next obfuscate the lookup tables by encoding their input and output. Encoding the input and output of a table T with bijective functions f_{in} and f_{out}, respectively, corresponds to replacing table T by $f_{out} \circ T \circ f_{in}^{-1}$. Hence, into the table, an input *de*coding and an output *en*coding are incorporated. To see that the application of encodings realizes obfuscation, observe that encoding the input of a lookup table changes the order of its rows and that encoding the output changes the value of the rows. The encodings are applied such that the operation of the overall implementation does not change. The collection of obfuscated lookup tables can be considered as the cryptographic key of the white-box implementation.

As indicated above, we propose to obfuscate the white-box implementation to protect against the attack of Billet et al. [3]. This attack extracts the AES key from a white-box AES implementation in a time complexity of 2^{30}. However, for the sake of simplicity and because an arbitrary set of obfuscation techniques can be applied, we present the white-box AES implementation and next Medusa without this additional layer of obfuscation.

We now give a brief treatment on the details of the white-box AES implementation of Chow et al. AES is a block cipher consisting of 10 rounds and with a 128-bit block size [11]. The 16 bytes of the 128-bit blocks on which AES operates are arranged into a 4×4 array of bytes. Such an array is called 'state'. If the state is denoted by a, then $a_{i,j}$ and (i, j) specify the byte in row i and column j. In this paper, indices are taken modulo 4.

AES encrypts and decrypts by applying transformations to its input state. The transformations are grouped into 10 rounds. A basic round consists of four transformations: *AddRoundKey*, *SubBytes*, *ShiftRows*, and *MixColumns*. The first 9 rounds of AES are basic rounds. The final round deviates from a basic round in that the *MixColumns* transformation is replaced by *AddRoundKey*.

The white-box AES implementation proposed by Chow et al. [7] implements each round as a sequence of table lookups and obfuscates the lookup tables by encoding their input and output. We first discuss the table-lookup implementation of AES that underlies the white-box implementation. As Medusa only operates on the white-box implementation of the basic rounds, we here focus on these rounds.

Instead of having *AddRoundKey* and *SubBytes* as two separate transformations, we can merge them into a single S-box operation by partial evaluation. Let $\kappa^r_{i,k}$ be byte (i,k) in the round key of round r, where, similarly as for the state, we refer to the 128-bit round key as a 4×4 array of bytes. For byte $a_{i,k}$ of the input state a of round r, the two transformations correspond to an application of the S-box $T^r_{i,k}$ that is defined by

$$T^r_{i,k}(a_{i,k}) = S(a_{i,k} \oplus \kappa^r_{i,k}).$$

We refer to these S-boxes as T-boxes.

Let M_j be the 32×32 bit matrix that implements the *MixColumns* transformation MC_j of column j, and let $M_{i,j}$ be the eight successive rows $8i, 8i+1, \ldots, 8(i+1)-1$ of M_j. The output $x_{i,k}$ of a T-box $T^r_{i,k}$ contributes to exactly one column j of the output of round r. The contribution is that $MC_{i,j}(x_{i,k})$ is XORed with output column j, where $MC_{i,j}(x_{i,k}) = x_{i,k} \cdot M_{i,j}$. If we implement both $MC_{i,j}(x_{i,k})$ and the XOR of two nibbles by lookup tables, then we can derive an implementation of round r that consists of lookup tables only. Fig. 2 depicts the computation of one output column in this implementation.

Chow et al. obfuscate the lookup-table implementation obtained by encoding the input and output of the tables. This is done by non-linear nibble encodings and linear mixing bijections. Non-linear nibble encodings are applied to the input and output nibbles of all the tables. The effect of these encodings is canceled out by choosing matching output and input encodings for successive tables in the lookup-table network. Linear mixing bijections are applied to the input and output of each table $MC_{i,j} \circ T^r_{i,k}$ in the following way. A mixing bijection MB^r_j encodes the output by multiplying it with

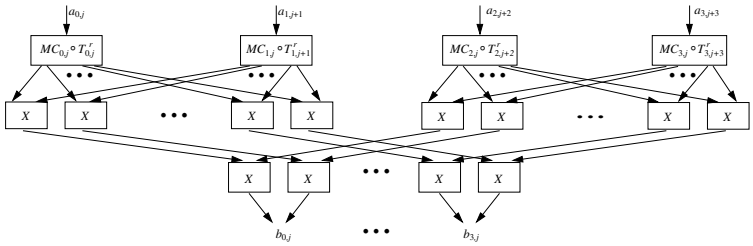


Figure 2: Network of lookup tables for computing the jth output column $(b_{0,j}, b_{1,j}, b_{2,j}, b_{3,j})$ of the rth AES round from input state a. Table X implements the XOR of two nibbles.

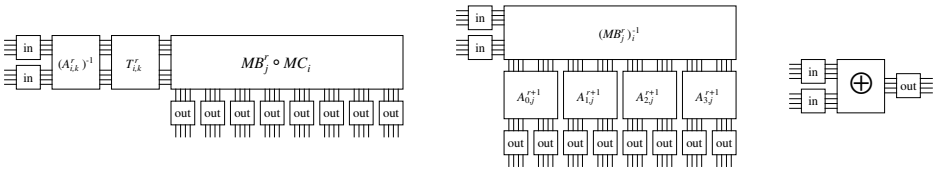


Figure 3: The Type II, III, and IV tables occur in a round of a white-box AES implementation.

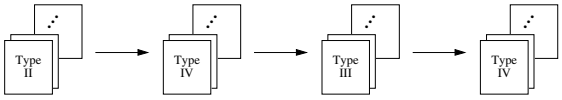


Figure 4: Overview of a single round in a white-box AES implementation.

a 32×32-bit invertible matrix, and a mixing bijection $A^r_{i,k}$ encodes the input by multiplying it with an 8×8-bit invertible matrix. The resulting lookup table, which is called a Type II table, is depicted in Fig. 3. To cancel out the effect of the mixing bijections introduced, additional lookup tables are included in the implementation. These lookup tables are called Type III tables; see Fig. 3. As for MC_j, the linear function $(MB^r_j)^{-1}$ is implemented by splitting it into four 8-bit to 32-bit linear functions $(MB^r_j)^{-1}_i$. An obfuscated XOR table is called a Type IV table. Fig. 4 depicts the overall structure of a round in the white-box AES implementation of Chow et al.

This completes the white-box implementation proposed by Chow et al. [7]. With respect to its performance, Chow et al. indicate that we lose a factor 10 in number of operations in comparison with a non-white-box implementation. The size penalty is more severe. The overall size increases from several kilobytes to 770,048 bytes. However, for many applications, such as in the PC domain, this size is still acceptable.

4. MEDUSA

Consider two white-box AES implementations that differ in the encodings applied and/or the keys that underly them. It can be verified that the lookup table networks associated with the two implementations only differ in the content of the lookup tables. The structures of the networks are the same. Hence, the only variability of a white-box AES implementation is in the content of its lookup tables. This implies that we can consider a white-box AES implementation as a cryptographic algorithm in its own right: its key is given by the collection of lookup tables and the cryptographic algorithm describes the employment of these tables, i.e., it describes for each table where to get its input from.

The idea of Medusa is to make binary program code tamper resistant by incorporating the code into the key of a white-box implementation, i.e., into the collection of lookup tables. This is done as follows. We partition the executable program code into fragments of 1024 bytes. Besides executable code, we interpret each fragment as a lookup table consisting of 256 rows of 4 bytes. The lookup table is included in the lookup table network that models the white-box implementation. By giving code an interpretation within the white-box implementation we make it tamper resistant since a change in the program code now implies an unintentional change in the key of the white-box implementation. Furthermore, we include the program code in the key in such a way that after changing the program repairing the key is complicated.

In its basic form, Medusa is only able to incorporate programs of limited size. In the full version of this paper we show how we can generalize the basic implementation such that programs of arbitrary size can be protected.

4.1 Medusa: a simplified version

Before we explain how Medusa includes code fragments into a white-box AES implementation, we show how we can include code fragments in the non-obfuscated lookup table network depicted in Fig. 2. This network, which implements the computation of a 32-bit output column b_j in an encryption round, simply XORs the four 32-bit values returned by the four lookup tables $MC_{i,j} \circ T^r_{i,k}$ with $i = 0,1,2,3$. Let $C^r_{i,k}$ with $i = 0,1,2,3$ be a lookup table that defines a 1024-byte code fragment $\hat{C}^r_{i,k}$. We define lookup table $W^r_{i,k}$, such that row x is given by $W^r_{i,k}(x) = MC_{i,j} \circ T^r_{i,k}(x) \oplus C^r_{i,k}(x)$. We then have that row x of lookup table $MC_{i,j} \circ T^r_{i,k}$ from Fig. 2 is given by $C^r_{i,k}(x) \oplus W^r_{i,k}(x)$.

We now make two copies N_0, N_1 of the network depicted in Fig. 2. In N_0 we replace each table $MC_{i,j} \circ T^r_{i,k}$ by $W^r_{i,k}$ and in N_1 we replace each table $MC_{i,j} \circ T^r_{i,k}$ by $C^r_{i,k}$. Network N_0 computes 32-bit value

$$x_0 = W^r_{0,j}(a_{0,j}) \oplus W^r_{1,j+1}(a_{1,j+1}) \oplus W^r_{2,j+2}(a_{2,j+2}) \oplus W^r_{3,j+3}(a_{3,j+3})$$

and network N_1 computes 32-bit value

$$x_1 = C^r_{0,j}(a_{0,j}) \oplus C^r_{1,j+1}(a_{1,j+1}) \oplus C^r_{2,j+2}(a_{2,j+2}) \oplus C^r_{3,j+3}(a_{3,j+3}).$$

Furthermore, $b_j = x_0 \oplus x_1$. Implementing the XOR of x_0 and x_1 via a network of XOR-tables X gives us a network $\mathcal{N}$ with the same functionality as the original network of Fig. 2. This network $\mathcal{N}$

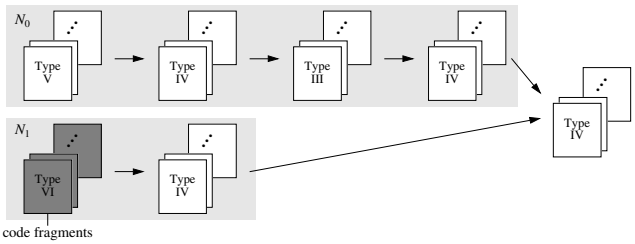


Figure 6: White-box implementation in which program code has been incorporated in the key.

has the property that four of its tables are given by program code. However, it is still not very complicated to change a single bit in the program code. To compensate for a bit flip in a table $C^r_{i,k}$, we only have to flip the bit that is at the same location in $W^r_{i,k}$.

4.2 Basic implementation of Medusa

Above we based Medusa on the non-encoded version of the white-box AES implementation. This non-encoded white-box implementation is given by the network depicted in Fig. 2. In Sect. 3 we transformed this network into a white-box AES implementation by applying non-linear nibble encodings and linear mixing bijections. More specifically, the linear encodings are such that the 8-bit input of a table $MC_{i,j} \circ T^r_{i,k}$ is encoded by an 8-bit mixing bijection $A^r_{i,k}$ and the 32-bit output is encoded by a 32-bit mixing bijection MB^r_j. To compensate for the linear encodings, the lookup table network is extended with tables that remove the effect of MB^r_j and that introduce the input encoding $A^{r+1}_{i,j}$ assumed in the next round. The network obtained is further obfuscated by non-linear nibble encodings.

A similar obfuscation is applied to the network $\mathcal{N}$ discussed above. To the tables $W^r_{i,k}$ in network N_0 we apply the same linear encodings $A^r_{i,k}$ and MB^r_j as to $MC_{i,j} \circ T^r_{i,k}$. Hence, table $W^r_{i,k}$ is transformed into

$$MB^r_j \circ W^r_{i,k} \circ (A^r_{i,k})^{-1}.$$

In the same way as for $MC_{i,j} \circ T^r_{i,k}$ we compensate for these linear encodings by including tables in $\mathcal{N}$.

The tables $C^r_{i,k}$ in network N_1 are encoded by the linear input encodings $A^r_{i,k}$ but not by the linear output encodings MB^r_j. Because we omit MB^r_j, we do not need to extend the table network of N_1 to compensate for it. The linear input encodings $A^{r+1}_{i,j}$ assumed in column j of the next round are incorporated in the table $C^r_{i,k}$. Let $A = (A^{r+1}_{0,j}, A^{r+1}_{1,j}, A^{r+1}_{2,j}, A^{r+1}_{3,j})$. Then this means that table $C^r_{i,k}$ is replaced by

$$A \circ C^r_{i,k} \circ (A^r_{i,k})^{-1}. \quad (1)$$

After applying non-linear nibble encodings to the network obtained, we get that the tables $W^r_{i,k}$ and $C^r_{i,k}$ are transformed into the Type V and Type VI tables depicted in Fig. 5. Fig. 6 gives an overview of the obfuscated version of $\mathcal{N}$.

By obfuscating a table $C^r_{i,k}$, we lose the property that it is equal to $\hat{C}^r_{i,k}$. To recover this property, we redefine the underlying table $C^r_{i,k}$, such that $C^r_{i,k}$ equals the code fragment $\hat{C}^r_{i,k}$ after the obfuscation instead of before the obfuscation. To formalize this, let the non-linear nibble encodings that are applied to the input and output of $C^r_{i,k}$ be given by $F = (F_0, F_1)$ and $G = (G_0, G_1, \ldots, G_7)$, respectively, where $F_l : 2^4 \to 2^4$ is the encoding of the lth input nibble and $G_l : 2^4 \to 2^4$ is the encoding of the lth output nibble. Then it follows from (1) that the obfuscated version of $C^r_{i,k}$ is given by

$$G \circ A \circ C^r_{i,k} \circ (A^r_{i,k})^{-1} \circ F^{-1}.$$

Hence, we get that the obfuscated version of $C^r_{i,k}$ is given by $\hat{C}^r_{i,k}$ if we define $C^r_{i,k}$ as

$$C^r_{i,k} = A^{-1} \circ G^{-1} \circ \hat{C}^r_{i,k} \circ F \circ A^r_{i,k}. \quad (2)$$

In other words, if we define $C^r_{i,k}$ as (2), then binary code fragment $\hat{C}^r_{i,k}$ is included in the white-box key in its original, unobfuscated form. This concludes the definition of the basic version of Medusa. The amount of program code that can be included in a white-box implementation by this basic version of Medusa is given by the total size of the Type VI tables, which is 16,384 per round and 147,456 bytes in total.

We now briefly discuss the performance penalty that is incurred by Medusa compared to a standard white-box implementation. It can be shown that if we apply Medusa to all 9 rounds, then the number of table lookups performed in an execution of the white-box implementation increases by 1,296. This is 9 per kilobyte of protected code. The increase in size is 294,912 bytes. Of this size overhead, 147,456 bytes are given by code. Hence, if we do not count the code that is protected as size overhead of the white-box implementation, the size overhead induced by Medusa is equal to the number of bytes that we want to protect. We emphasize that the program code that is protected is not affected by Medusa and thus neither is its execution time.

To put Medusa to practice, we have developed an automatic tool. The input to the tool is the binary B of the software that has to be protected and an AES key K. In linear time in $|B|$, the tool derives a white-box implementation with a white-box key that contains B and which implements AES with key K. If desired the algorithmic part of the white-box implementation is also included in the white-box key. The binary B is activated by a jump or jump to subroutine instruction, where the argument of the instruction is an address in the table.

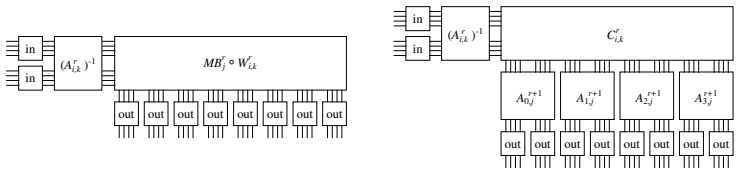
5. SECURITY OF MEDUSA

In this section we discuss the security of Medusa, where we assume the white-box attack model. As mentioned in the introduction, this attack model is the most realistic one for software protection.

Because Medusa only checks the integrity of software during encryption/decryption, it is essential for the system to which Medusa is applied that encryption/decryption plays an important role. Correspondingly, we assume in this section that it is the goal of an attacker to modify the software protected by Medusa without losing the ability to decrypt/encrypt properly. Furthermore, the modifications made in the software have to be such that they result in an intended change in functionality. Changing the software to a meaningless bitstring is not considered to be a successful attack.

In Sect. 5.1 5.2, and 5.3 we discuss several strategies for an attacker to attack Medusa. We categorize them into three types. In the first type of attack an attacker accepts incorrect encryptions or decryptions. In the second type of attack an attacker tries to realize a modification of protected code by repairing a white-box key after it has been invalidated by the modification. In the third type of attack an attacker tries to break the dual interpretation that has been given to code.

From the security discussion in this section it will follow that, besides its strengths, Medusa also has its limitations. However, as mentioned in the introduction, the software tamper-resistance



Type V Type VI

Figure 5: The Type V and Type VI tables that are used in Medusa and not in a standard white-box AES implementation.

problem is tackled in practice by applying multiple complementary tools instead of only one. Medusa is an effective new tool for the toolbox.

5.1 Accepting incorrect encryptions and decryptions

The idea of Medusa is that by including code in a white-box implementation, the integrity of the code is checked each time that the white-box implementation is executed. However, it is not the case that the integrity of each individual bit of protected code is checked each time the white-box implementation is executed. A bit of a lookup table $\hat{C}^r_{i,k}$ is only checked if the table row containing it is accessed, and $\hat{C}^r_{i,k}$ contains 256 rows of which exactly one is accessed during an execution of the white-box implementation. Hence, if we would apply the white-box implementation to random data blocks, the probability that a bit flip is detected in a single run of the white-box implementation is only $1/256$. This means that if an attacker changes only one bit, then, on average, 256 encryptions/decryptions are required before the incorrect white-box implementation also results in an incorrect operation. This number will decrease considerably for changes of the software that affect multiple table rows. We note that the consequences of an incorrect operation of a white-box implementation can be made more serious by choosing a block-cipher mode in which an encryption (decryption) error propagates to further encryptions (decryptions).

5.2 Repairing a white-box key

As second type of attack we mentioned that an attacker may try to realize a modification of protected code by repairing a white-box key after it has been invalidated by a modification. In Sect. 4 we saw that an attack of this type is successful for the naive implementation of Medusa where code tables are included in the non-obfuscated network of Fig. 2. We indicated that we can compensate for a bit flip in a table $C^r_{i,k} = \hat{C}^r_{i,k}$ by flipping the bit that is at the same location in $W^r_{i,k}$.

By encoding the output of each table $W^r_{i,k}$ with a mixing bijection MB^r_j and by omitting this linear output encoding for each table $C^r_{i,k}$, we have made it more complicated to repair the white-box key after flipping a bit in a $\hat{C}^r_{i,k}$. It no longer suffices to flip a single bit in $W^r_{i,k}$, but we have to change the value of a complete row in $W^r_{i,k}$. This follows from the following theorem, where we prove the relation between a row in code $\hat{C}^r_{i,k}$ and the corresponding row in $W^r_{i,k}$.

THEOREM 1. *Suppose that an attacker wants to XOR a row of $\hat{C}^r_{i,k}$ with a value Δ. Then the invalidated key can be repaired by XORing the corresponding row in the obfuscated version of table $W^r_{i,k}$ with the 32-bit value*

$$H \circ MB^r_j \circ A^{-1} \circ G^{-1}(\Delta), \tag{3}$$

where $A = (A^{r+1}_{0,j}, A^{r+1}_{1,j}, A^{r+1}_{2,j}, A^{r+1}_{3,j})$ and $G = (G_0, G_1, \ldots, G_7)$ are the linear encoding and nibble encoding applied to the output of $C^r_{i,k}$ and MB^r_j and $H = (H_0, H_1, \ldots, H_7)$ are the linear encoding and nibble encoding applied to the output of $W^r_{i,k}$.

PROOF. From (2) it follows that as a result of change Δ in $\hat{C}^r_{i,k}$, the corresponding row p of $C^r_{i,k}$ is XOR-ed with the value $A^{-1} \circ G^{-1}(\Delta)$. To compensate for this change, the value $A^{-1} \circ G^{-1}(\Delta)$ also has to be XORed with row p of $W^r_{i,k}$. However, the attacker does not have $W^r_{i,k}$ to his disposal, but only its obfuscated version. That is, a mixing bijection MB^r_j and non-linear nibble encodings H are applied to row p of $W^r_{i,k}$. As a result, an attacker has to XOR row p of $W^r_{i,k}$ with (3), which was to be proved. □

A straightforward approach for finding the 32-bit value (3) is by trial-and-error. Note that the linear and nibble encodings in (3) are unknown to an attacker. In a trial-and-error approach, an attacker only needs a very limited knowledge of the white-box implementation. He only has to locate the row in $W^r_{i,k}$ that has to be changed and not, for instance, the precise content of one or more tables. This is an advantage for the attacker as we proposed to obfuscate a white-box implementation by intertwining the lookup tables by applying code obfuscation techniques. Obviously, the time-complexity of performing this trial-and-error attack is 2^{32}. We now show how this complexity can be increased.

A round of the white-box AES implementation consists of four lookup table networks. Each of these four networks derives from four input bytes a single output column, which also consists of four bytes. We can increase the 2^{32} complexity by merging the computation of multiple columns into a single lookup table network. This means that we either derive a single network that derives all four output columns from all 16 input bytes, or we derive two networks in which one network derives the first two columns from the eight corresponding input bytes and the other network derives the last two columns from the other eight input bytes. The first option results in a complexity of 2^{128} for the trial-and-error attack discussed and the latter option results in a complexity of 2^{64} for this attack. However, an increase in complexity is at the cost of an increase in size and execution time of the white-box implementation. Hence, a trade-off has to be made between security of the protected program and performance of the cryptographic algorithm. We here indicate how we can increase the complexity from 2^{32} to 2^{64}. The 2^{128} complexity can be obtained similarly.

The white-box AES implementation is based on the lookup table implementation depicted in Fig. 2. A table $MC_{i,j} \circ T^r_{i,k}$ defines the contribution of input byte $a_{i,k}$ to an output column b_j. We redefine table $MC_{i,j} \circ T^r_{i,k}$, such that it defines the contribution of $a_{i,k}$ to a pair b_j, b_{j+1} of output columns. The resulting table is denoted by $\mathcal{T}^r_{i,k}$. If $a_{i,k}$ contributes to b_j, then the row of table $\mathcal{T}^r_{i,k}$ consists of 64 bits, the first 32 bits of which are given by the corresponding row in $MC_{i,j} \circ T^r_{i,k}$ and the last 32 bits are given by zeros. If, on the other hand, the input byte $a_{i,k}$ contributes to column b_{j+1}, then the order of the 32 zeros and the value of the corresponding row in $MC_i \circ T^r_{i,k}$ are swapped. The output columns b_j, b_{j+1} can now

be obtained by XORing the 64-bit values returned by their eight corresponding tables.

We obfuscate the resulting network in the same way as before. However, we now use a 64×64-bit mixing bijection MB^r_j to encode the rows of $\mathcal{T}^r_{i,k}$. If we next apply Medusa, the table $\mathcal{T}^r_{i,k}$ is split into two lookup tables that, as $\mathcal{T}^r_{i,k}$, consist of 256 rows of 64 bits. After obfuscation, one of these two tables represents a code fragment.

It can be verified that Theorem 1 also holds for this implementation of Medusa, where $H, MB^r_j, A,$ and G are now 64-bit mappings. Hence, if an attacker wants to add a value Δ to a row of $\hat{C}^r_{i,k}$, then the complexity of repairing the white-box key by performing a proposed trial-and-error approach to find the proper change for table $W^r_{i,k}$ is 2^{64}. The increase in complexity is at the cost of a performance penalty. We indicated in Sect. 4 that in a basic Medusa implementation the additional number of table lookups that is performed in comparison with a standard white-box AES implementation is 9 per kilobyte of protected code. By going to a complexity of 2^{64}, the additional number of table lookups increases from 9 to 16.5 per kilobyte of protected code. The size penalty of Medusa increases from being equal to the number of bytes that we protect to 3 times the number of bytes that we protect, where we do not count code as size overhead in a white-box implementation.

Above we discussed the complexity of achieving a software tampering if an attacker repairs a white-box key by simply checking all 32, 64, or 128-bit values to find the value (3). More efficient approaches exist for finding the value (3). These approaches require an attacker to determine a substantial part of the lookup-table network in which the modified table $\hat{C}^r_{i,k}$ is contained. For the basic implementation of Medusa this means that the attacker has to determine a substantial part of the network depicted in Fig. 6. However, we consider this to be hard as we assumed the white-box implementation to be obfuscated in order to protect against the attack of Billet et al. [3]. We leave the discussion of other approaches that try to find the value (3) by other strategies than trial-and-error outside the scope of this paper.

5.3 Breaking the dual interpretation

As a third type of attack we mentioned attacks that try to break the dual interpretation of protected code. As a first attack of this type we consider one that tries to break the dual interpretation of a code table by removing it from a white-box implementation. That is, an attacker removes a code table and compensates this in the associated table $W^r_{i,k}$. Note that removing all code tables corresponds to going back to the standard white-box implementation to which we applied Medusa. Compensating the removal of a code table $\hat{C}^r_{i,k}$ in a table $W^r_{i,k}$ means XORing $C^r_{i,k}$ and $W^r_{i,k}$. Above we showed that repairing a key is difficult after XORing a row of a code table $\hat{C}^r_{i,k}$ with a value Δ. The reason is that it is difficult to find the value (3) that we have to XOR with the corresponding row in the obfuscated version of $W^r_{i,k}$. It can be verified that to XOR tables $C^r_{i,k}$ and $W^r_{i,k}$, we have to XOR the obfuscated version of $W^r_{i,k}$ with the table

$$H \circ MB^r_j \circ A^{-1} \circ G^{-1}(\hat{C}^r_{i,k}),$$

where we use the same notation as in Theorem 1. Similarly as for finding the value (3), it can be argued that to derive this table an attacker either has to perform a trial-and-error approach with a complexity that can increased to a value as large as 2^{128}, or the attacker must be able to remove the obfuscation put on the white-box implementation.

An alternative approach to break the dual interpretation of protected code is to use a 'shadow image'. Suppose that an attacker has enough free storage to his disposal to make a shadow image of the protected software. Van Oorschot, Somayaji, and Wurster [18] show that by modifying the kernel, calls to instructions that are used as data can be redirected to the shadow image. The original code image can subsequently be changed without affecting the use of it as data, which is done to validate the integrity of the software. As a solution to this problem Giffin, Christodorescu, and Kruger [12] propose to let software modify itself. An alternative solution is to dispose an attacker of the possibility to enlarge the software program. This prerequisite is, for instance, satisfied is we have limited storage.

This attack exploits a vulnerability that is shared by all techniques that protect the integrity of software by including into the software some kind of self-check. To see this, observe that in a self-check program code is interpreted as data if it is is used in an integrity check and that it is interpreted as code, otherwise. Hence, the attack is not only a threat to Medusa, but also, for instance, to the techniques that are based on hashing (see Sect. 2).

Instead of modifying the kernel, we can also redirect calls to instructions that are used as data by implementing this redirection in the code that contains the calls. That is, in a white-box implementation calls to data are replaced by calls to the shadow image. This attack can be optimized such that we do not have to copy the complete protected software program, but only parts of it that are changed. Observe that the amount the software that has to be changed increases if we include into the white-box key the algorithmic part of the white-box implementation that specifies the employment of the tables. The same solutions as presented for the kernel-based attack are useful against this attack.

6. CONCLUSION

Until now, the value of a white-box implementation has only been in its ability to hide a cryptographic key. In this paper we give a practical technique that uses a white-box implementation to make code tamper resistant. Our technique works by giving code a dual interpretation. It is executable code and key in a white-box implementation of a block cipher at the same time. This makes code tamper resistant as changing the code implies that we invalidate the white-box key. Furthermore, we indicated that it is difficult for an attacker to repair an invalidated white-box implementation. Our technique has the advantage that the protected code need not be modified. In this paper we only discussed a basic implementation that is able to protect programs of limited size. In the full paper, we show how to extend the implementation to protect programs of arbitrary size.

7. REFERENCES

[1] D. Aucsmith. Tamper resistant software: an implementation. In *Proceedings of the 1st International Workshop on Information Hiding*, pages 317–333, 1996.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of Crypto 2001*, pages 1–18, 2001.

[3] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white-box AES implementation. In *Proceedings of the 11th Annual Workshop on Selected Areas in Cryptography*, pages 227–240, 2004.

[4] B. Blietz and A. Tyagi. Software tamper resistance through dynamic program monitoring. In *Proceedings of the 1st International Conference on Digital Rights Management: Technology, Issues, Challenges and Systems*, 2005.

[5] H. Chang and M. Atallah. Protecting software code by guards. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, pages 160–175, 2001.

[6] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. In *Proceedings of the 5th International Workshop on Information Hiding*, pages 400–414, 2002.

[7] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. White-box cryptography and an AES implementation. In *Proceedings of the 9th Annual Workshop on Selected Areas in Cryptography*, pages 250–270, 2002.

[8] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. A white-box DES implementation for DRM applications. In *Proceedings of the 2nd ACM Workshop on Digital Rights Management*, pages 1–15, 2002.

[9] F. Cohen. Operating system protection through program evolution. *Computers and Security*, pages 565–584, 1993.

[10] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report no. 148, Department of Computer Science, University of Auckland, 1997.

[11] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer, 2002.

[12] J. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 23–32, 2005.

[13] L. Goubin, J. Masereel, and M. Quisquater. Cryptanalysis of white box DES implementations. To appear in *Proceedings of the 14th Annual Workshop on Selected Areas in Cryptography*, 2007.

[14] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Proceedings of the 1st ACM Workshop on Digital Rights Management*, pages 141–159, 2001.

[15] M. Jacob, D. Boneh, and E. Felten. Attacking an obfuscated cipher by injecting faults. In *Proceedings of the 2nd ACM Workshop on Digital Rights Management*, pages 16–31, 2002.

[16] H. Link and W. Neumann. Clarifying obfuscation: improving the security of white-box DES. In *Proceedings of the International Symposium on Information Technology: Coding and Computing*, pages 679–684, 2005.

[17] B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *Proceedings of Eurocrypt 2004*, pages 20–39, 2004.

[18] P. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, pages 82–92, 2005.

[19] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: obstructing static analysis of programs. Technical Report no. CS-2000-12, Department of Computer Science, University of Virginia, 2000.

[20] P. Wang, S. Kang, and K. Kim. Tamper resistant software through dynamic integrity checking. In *Proceedings of the 2005 Symposium on Cryptography and Information Security*, 2005.

[21] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Poland, 2002.

[22] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. To appear in *Proceedings of the 14th Annual Workshop on Selected Areas in Cryptography*, 2007.

[23] G. Yuval. Reinventing the travois: encryption/MAC in 30 ROM bytes. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, pages 205–209, 1997.