White-box Implementation of Stream Cipher

Kazuhide Fukushima, Seira Hidano and Shinsaku Kiyomoto
KDDI Research, Inc., 2-1-15 Ohara, 356-8502, Fujimino-shi, Saitama, Japan
{*ka-fukushima, se-hidano, kiyomoto*}*@kddilabs.jp*

Keywords: White-box Cryptography, Software Security, Tamper-resistant Software, Space Hard Cipher, Stream Cipher, KCipher-2.

Abstract: White-box cryptography is a software obfuscation technique for cryptography implementation. It can protect the secret key even if an attacker has full access and control over the cryptosystem implementation and its execution platform. There have been several proposals for a white-box implementation of cryptography. We propose a white-box implementation of a stream cipher that can achieve the same asymptotic performance as the standard implementation. Our black-box implementation of KCipher-2 achieves low storage consumption of no more than 2 megabytes and is suitable for a PC, tablet, and smartphone. On the other hand, we can achieve *space hard* implementation to protect against a code-lifting attack. Furthermore, the implementation can protect against black-box attacks and a BGE attack.

1 INTRODUCTION

The conventional threats to a secret key cryptosystem are considered in a black-box model. In this model, an attacker cannot obtain the information on the cryptosystem implementation. Thus, the attacker can obtain no information on the cryptographic process as long as the secret key is securely managed in the endpoints. Kocher (Kocher, 1996) has proposed a side-channel attack against implementation of Diffie-Hellman, RSA, and DSS, and this has led to the concept of a grey-box model. By means of this model, an attacker can obtain side-channel information on the execution process of the cryptosystem such as power consumption, electromagnetic radiation, and execution time. Chow et al. (Chow et al., 2003a; Chow et al., 2003b) have proposed white-box implementations of AES and DES. These implementations assume a white-box model where an attacker has full access and control over the implementation and its execution platform.

Software obfuscation transforms a program into an obfuscated one, which is hard to analyze while preserving its functionality. Practical software obfuscation techniques can be classified into two categories: data obfuscation and control obfuscation. Data obfuscation changes the data structures in software. Collberg et al. (Collberg et al., 1997) proposed *change encoding* that encodes variables with a linear function. The data obfuscation techniques can transform the secret key embedded in software. Control obfuscation changes the control flow in software. The control obfuscation technique can protect the secret key against static analysis. For example, we transform data in software to a function and apply a control obfuscation technique to protect the function.

However, Barak et al. (Barak et al., 2001) showed the existence of functions that cannot be obfuscated. They prove that no obfuscator satisfies the *virtual black box* property that is, there is no obfuscator O, such that for any program P, anything that can be efficiently computed from $O(P)$ can be efficiently computed given oracle access to P. Subsequently, the impossibility of software obfuscation have been published (Goldwasser and Kalai, 2005; Goldwasser and Rothblum, 2007; Hofheinz et al., 2007; Hada and Sakurai, 2007), and it is difficult for software implementations achieve black-box security.

Security based on the white-box model is, therefore, desirable for various services in the real world. A digital rights management (DRM) system allows a legitimate user to use content. The media player for the authorized user embeds the secret key for the user. However, the DRM system will be broken if an attacker can extract the secret key from the cryptosystem implementation in the media player. The attacker can use various reverse-engineering tools such as a network monitoring tool, disassembler, debugger, emulator and hexadecimal editor.

In this paper, we propose a white-box implemen-

Fukushima, K., Hidano, S. and Kiyomoto, S.
White-box Implementation of Stream Cipher.
DOI: 10.5220/0006115202630269
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 263-269
ISBN: 978-989-758-209-7

tation of a stream cipher. Our implementation can achieve the same performance as keystream output of a standard implementation while not imposing an unreasonable level of overhead on the key initialization process.

2 RELATED WORK

White-box Cryptography and Attacks. Chow et al. devised the concept of white-box cryptography, and they proposed a white-box implementation of AES (Chow et al., 2003b) and DES (Chow et al., 2003a). Billet et al. (Billet et al., 2005; Muir, 2013) proposed a BGE attack against the white-box implementation of AES by Chow et al. The BGE attack extracts the entire AES secret key embedded in the white-box implementation.

Xiao and Lai (Xiao and Lai, 2009) proposed a new white-box AES implementation which is claimed to be resistant to a BGE attack. Mulder et al. (Mulder et al., 2013) showed that a practical cryptanalysis can be applied to their white-box implementation. Their attack uses a modified variant of the linear equivalence algorithm.

Bringer et al. (Bringer et al., 2006a) proposed a way to conceal the algebraic structure of a traceable block cipher by adding perturbations to its description. Then, they proposed a white-box implementation for AEw/oS (Bringer et al., 2006b) that is a variant of the AES block cipher where the S-boxes are part of the key and unknown to the adversary. Mulder (Mulder et al., 2010) developed a cryptanalysis of the perturbated white-box AEw/oS implementation that can be extended to perturbated white-box AES implementations.

Karroumi (Karroumi, 2010) proposed new white-box implementation based on dual ciphers of AES. However, Klinec (Klinec, 2013) showed that this implementation can be broken with the same time complexity as the white-box implementation of AES. Bos et al. (Bos et al., 2015) demonstrated that publicly available white-box implementations cannot protect against the differential computation analysis (DCA) technique and the attack can extract the secret key within seconds.

White-box Implementation Towards Probable Security. Sasdrich et al. (Sasdrich et al., 2016) proposed a white-box implementation of AES on reconfigurable hardware assuming a gray-box attacker who can execute side-channel analysis including a differential computational analysis attack and differential power analysis attack. Cho et al. (Cho et al., 2016) proposes a new class of encryption scheme, hybrid WBC, that can protect against their threat model and achieve reasonable performance 1.3 times slower than AES.

Biryukov et al. (Biryukov et al., 2014) designed several encryption schemes based on the ASASA structure ranging based on general affine transformations combined with specially designed low degree non-linear layers. Bogdanov and Isobe (Bogdanov and Isobe, 2015) proposed a family of white-box secure block ciphers SPACE and introduced the notion of *space hardness* to protect against a code-lifting attack.

Saxena et al. (Saxena et al., 2009) defined security notions for white-box cryptography. Fouque et al. (Fouque et al., 2016) proposed white-box primitives offering provable security guarantees and concrete instantiations for their construction with low overhead.

3 KCipher-2

KCipher-2 was proposed by Kiyomoto *et al.* (Kiyomoto et al., 2007) and has a similar structure to SNOW-family stream ciphers. KCipher-2 consists of two feedback shift registers (FSRs) *FSR-A* and *FSR-B*, and a non-linear function with four internal registers as shown in Fig. 1. The dotted lines show the paths of one-bit variables which determine a multiplication.

3.1 Non-linear Function

The non-linear function of KCipher-2 has internal registers $R1$, $R2$, $L1$, $L2$. The values of two registers of FSR-A and four registers of FSR-B are fed into the non-linear function, and a 64-bit keystream is generated from a non-linear function every cycle. Figure 1 shows the non-linear function of KCipher-2. The non-linear function includes four substitution steps that are indicated by *Sub*.

The *Sub* step divides the 32-bit input string into four 1-byte strings and applies a non-linear permutation to each byte using an 8-to-8 bit substitution, and then applies a 32-to-32 bit linear permutation. The 8-to-8 bit substitution is the same as s-boxes of AES (Daemen and Rijmen, 2002), and the permutation is the same as the AES *MixColumn* operation.

3.2 Keystream Output

Let keystream at time t be $Z^{(t)} = (ZH^{(t)}, ZL^{(t)})$, where $ZH^{(t)}$ is a higher 32-bit keystream, and $ZL^{(t)}$ is a

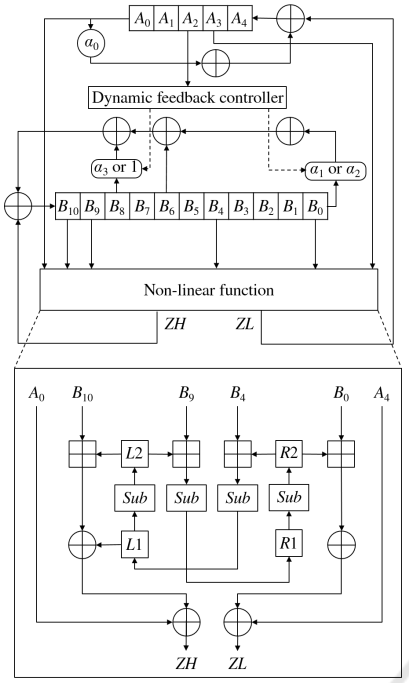

Figure 1: KCipher-2 Stream Cipher.

lower 32-bit keystream. The keystream is calculated as follows:

$$ZL^{(t)} = B_0^{(t)} \boxplus R2^{(t)} \oplus R1^{(t)} \oplus A_4^{(t)},$$
$$ZH^{(t)} = B_{10}^{(t)} \boxplus L2^{(t)} \oplus L1^{(t)} \oplus A_0^{(t)},$$

where $A_x^{(t)}$ and $B_x^{(t)}$ denote outputs of FSR-A and FSR-B at position x and time t and $R1^{(t)}$, $R2^{(t)}$, $L1^{(t)}$, and $L2^{(t)}$ denote the internal registers at time t. The symbol $\oplus$ denotes a 32-bit exclusive-or and $\boxplus$ denotes a 32-bit addition. Finally, the internal registers are updated as follows:

$$R1^{(t+1)} = Sub(L2^{(t)} \boxplus B_9^{(t)}), \quad R2^{(t+1)} = Sub(R1^{(t)}),$$
$$L1^{(t+1)} = Sub(R2^{(t)} \boxplus B_4^{(t)}), \quad L2^{(t+1)} = Sub(L1^{(t)}).$$

3.3 Key Initialization Process

The initial internal state is generated from a 128-bit initial key and a 128-bit initial vector (IV) using the key scheduling algorithm. The key scheduling algorithm is similar to the round key generation function of AES, and the algorithm extends the 128-bit initial key to 384 bits. The key scheduling algorithm is described as:

$$K_i = \begin{cases} IK_i & (0 \le i \le 3) \\ K_{i-4} \oplus Sub((K_{i-1} \lll 8) \oplus \\ \quad (K_{i-1} \ggg 24)) \oplus Rcon[i/4-1] & (i = 4n) \\ K_{i-4} \oplus K_{i-1} & (i \neq 4n) \end{cases},$$

where $IK = (IK_0, IK_1, IK_2, IK_3)$ is the initial key and $0 \le i \le 11$. The function *Sub* in the key scheduling algorithm is the same as that in the non-linear function. This function is different from the round key generation function of AES, while the other part of the key scheduling algorithm is the same as the AES round key generation. *Rcon*[i] denotes the round constant word array in AES. The internal state is initialized with K_i and $IV = (IV_0, IV_1, IV_2, IV_3)$ as follows;

$$\begin{aligned} &A_0 = K_4, \quad A_1 = K_3, \quad A_2 = K_2, \quad A_3 = K_1, \\ &A_4 = K_0, \quad B_0 = K_{10}, \quad B_1 = K_{11}, \quad B_2 = IV_0, \\ &B_3 = IV_1, \quad B_4 = K_8, \quad B_5 = K_9, \quad B_6 = IV_2, \\ &B_7 = IV_3, \quad B_8 = K_7, \quad B_9 = K_5, \quad B_{10} = K_6. \end{aligned}$$

The internal registers, $R1$, $R2$, $L1$, and $L2$ are set to 0. After the above processes, the cipher clocks 24 times $(1 \le t \le 24)$, updating the internal states as follows;

$$A_i^{(t+1)} = \begin{cases} A_{i+1}^{(t)} & (0 \le i \le 3) \\ \alpha_0 \cdot A_0^{(t)} \oplus A_3^{(t)} \oplus ZL^{(t)} & (i = 4) \end{cases},$$

$$B_i^{(t+1)} = \begin{cases} B_{i+1}^{(t)} & (0 \le i \le 9) \\ f(A_2^{(t)}) \cdot B_0^{(t)} \oplus B_1^{(t)} \oplus B_6^{(t)} & \\ \quad \oplus g(A_2^{(t)}) \cdot B_8^{(t)} \oplus ZH^{(t)} & (i = 10) \end{cases}.$$

The f and g are dynamic feedback controller functions defined as;

$$f(A_2^{(t)}) = \begin{cases} \alpha_1 & (A_2^{(t)}[30] = 1) \\ \alpha_2 & (A_2^{(t)}[30] = 0) \end{cases},$$

$$g(A_2^{(t)}) = \begin{cases} \alpha_3 & (A_2^{(t)}[31] = 1) \\ 1 & (A_2^{(t)}[31] = 0) \end{cases},$$

and α_0, α_1, α_2, α_3 are constants in $GF(2^{32})$.

4 PROPOSED METHOD

We propose a white-box implementation for stream ciphers. Figure 2 shows the overview. The white-box implementation calculates initialized internal states using look-up tables (LUTs) in which the secret key is embedded. Once the implementation completes the key initialization process, it outputs keystream in exactly the same way as the standard implementation. Thus, our implementation imposes no overhead on the keystream output process.

An attacker can obtain the initialized internal states in the white-box implementation. The key initialization process must be noninvertible so that the attacker cannot calculate the key and IV from the initialized internal states.

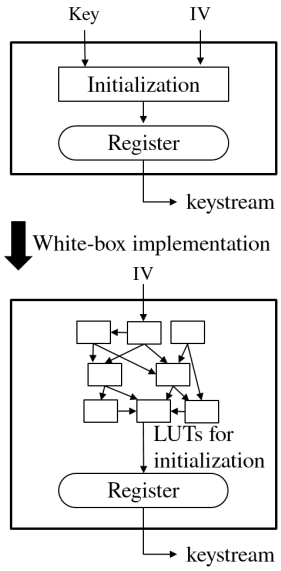

Figure 2: White-Box Implementation of Stream Cipher.

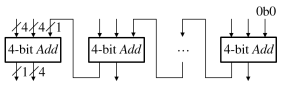

Figure 3: 32-bit Addition.

We show the white-box implementation of KCipher-2. The key initialization process of KCipher-2 is proved to be noninvertible. We can thus apply our proposed implementation to KCipher-2. Section 4.1 shows that all of the basic operations in the initialized process in KCipher-2 can be implemented using LUTs. We show the entire white-box implementation of KCipher-2 in Sect. 4.2.

4.1 Basic Operations

The key initialization process of KCipher-2 uses 32-bit exclusive-or, 32-bit addition, multiplication over $GF(2^{32})$, 32-bit *Sub* function and dynamic feedback controller function f and g. We implement these operations using four-bitwise operation based on LUTs. The dynamic feedback controller functions can be implemented using multiplication over $GF(2^{32})$ and a 32-bit multiplexer.

32-bit Exclusive-OR. The 32-bit exclusive-or can be implemented using eight 8-bit to 4-bit LUTs.

32-bit Addition. The 32-bit addition can be implemented using eight LUTs for 4-bit addition. The LUT has 9-bit input and 5-bit output to handle carries. Figure 3 shows the structure of the 32-bit addition in detail.

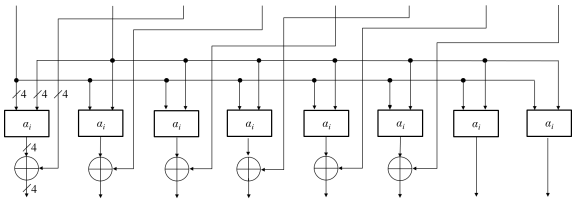

Figure 4: Multiplication over $GF(2^{32})$.

Multiplication Over GF(2^{32}). The multiplication of constant $\alpha_i \in GF(2^{32})$ and $x \in GF(2^{32})$ can be calculated with table lookup and exclusive-or operations as;

$$\alpha_i \cdot x = (x \ll 8) \oplus \texttt{alpha\_i}[(x \gg 24)],$$

where $\ll$ and $\gg$ is left-shift and right-shift operation respectively. The table `alpha_i` is a 32-bit to 32-bit LUT; however, we divide it into eight 4-bit to 4-bit LUTs. Figure 4 shows the structure of the multiplication over $GF(2^{32})$ in detail.

***Sub* Function.** The *Sub* function can be implemented using four 8-bit to 32-bit LUTs and three 32-bit exclusive-or operations as;

$$\begin{aligned} Sub(x) = &\texttt{sub\_0}[x\&\texttt{0xFF}] \oplus \texttt{sub\_1}[(x \gg 8)\&\texttt{0xFF}] \\ &\oplus \texttt{sub\_2}[(x \gg 16)\&\texttt{0xFF}] \\ &\oplus \texttt{sub\_3}[(x \gg 24)\&\texttt{0xFF}], \end{aligned}$$

where & is logical AND operation. Figure 5 shows the detailed structure of the *Sub* function.

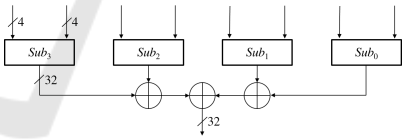

Figure 5: *Sub* Function.

32-bit Multiplexer. The dynamic feedback controller functions f and g can be implemented using 32-bit multiplexers. The function f outputs one of $\alpha_1 \cdot B_0$ and $\alpha_2 \cdot B_0$ according to $A_2^{(t)}[30]$, and g outputs one of $\alpha_3 \cdot B_8$ and B_8 according to $A_2^{(t)}[31]$ and the multiplexer extracts the appropriate output. The 32-bit multiplexer can be implemented with eight 12-bit to 4-bit LUTs. Figure 6 shows the structure of the 32-bit multiplexer in detail.

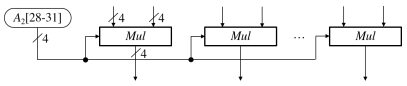

Figure 6: 32-bit Multiplexer.

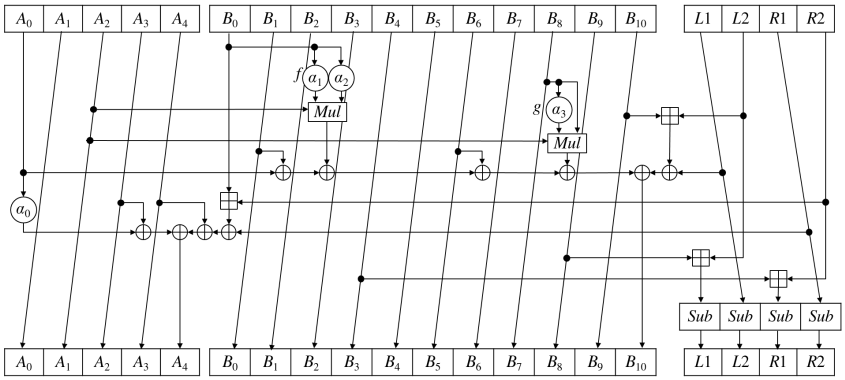

Figure 7: One Cycle of Key Initialization Process of KCipher-2.

4.2 Implementation of KCipher-2

The key initialization process of KCipher-2 based on LUTs is shown in Fig. 7. The figure shows one cycle of the key initialization process. The entire key initialization process consists of 24 repeats of the cycle.

5 EVALUATION

We evaluate the storage size required for the white-box implementation of KCipher-2. We then analyze the security of the implementation.

5.1 Storage Size

We estimate the total size of LUTs used in the white-box implementation of KCipher-2. First, we show the size of LUTs for basic operations. Then, we evaluate the total storage size for the one cycle and the entire key initialization process.

The size of the LUT is estimated by $n_b \cdot 2^{n_a}$ bits or $n_b \cdot 2^{n_a-3}$ bytes where n_a is the input bit length and n_b is the output bit length. Table 1 shows the storage size of LUTs for the basic operations used in the white-box implementation of KCipher-2. One cycle of the key initialization process contains ten 32-bit exclusive-or operations, four 32-bit additions, four multiplications over $GF(2^{32})$, four *Sub* functions and two 32-bit multiplexers. Thus, the storage size required for one cycle of the key initialization process is 87 kilobytes. The total storage size for the entire key initialization process is 87×24 kilobytes or 2,088 kilobytes.

Optimization with Pre-computation. We can reduce the storage size of our white-box implementation. The input of some LUTs is independent of the IV; that is, it depends only on the embedded secret key. These LUTs can be removed using a pre-computation technique. This optimization facilitates a 245-kilobyte reduction, and the total storage size is 1,843 kilobytes.

5.2 Security

We consider the security of our white-box implementation against a black-box attack and BGE attack.

Black-box Attack. A black-box attack can be protected against using the input/output encoding technique proposed by Chow et al. (Chow et al., 2003b). An attacker tries to extract some parts of the key from the input/output of LUTs. For example, $Sub(K_5 \boxplus 0)$ is assigned to $L1$ and $Sub(K_8 \boxplus 0)$ is assigned to $R1$ at the first cycle of the key initialization process. The attacker can extract K_5 and K_8 from the input/output of the LUTs for 32-bit addition.

To protect a LUT T, we choose bijections P and Q, and generate the new table $T' = Q \circ T \circ P^{-1}$. If the output of T feeds into another table U, we have to choose the bijections so that output encoding of T and input encoding U cancel each other. Table T and U can be protected as follows;

$$\begin{aligned} U' \circ T' &= (R \circ U \circ Q^{-1}) \circ (Q \circ T \circ P^{-1}) \\ &= R \circ (U \circ T) \circ P^{-1}, \end{aligned}$$

where $T' = Q \circ T \circ P^{-1}$ and $U' = R \circ U \circ Q^{-1}$.

The carry of 4-bit addition can also be protected by the input/output encoding technique. Some 4-bit addition LUTs output flipped carry and the others output unflipped carry. The total number of 4-bit additions is 384 in the white-box implementation and 377 in the optimized implementation. Thus, the computational complexity to guess the input/output encoding of the entire key initialization process is 2^{384} and 2^{377}, respectively.

Table 1: Storage Size of LUTs for Basic Operations.

Operation	Structure	Storage Size
32-bit Exclusive-OR	8-bit to 4-bit LUTs × 8	1 [KB]
32-bit Addition	9-bit to 5-bit LUTs × 8	2.5 [KB]
Multiplication over $GF(2^{32})$	8-bit to 4-bit LUTs × 8 + 4-bit Exclusive-OR × 6	1.75[kB]
Sub Function	8-bit to 32-bit LUTs × 4 + 32-bit Exclusive-OR × 3	7[KB]
32-bit Multiplexer	12-bit to 4-bit LUTs × 8	16[KB]

BGE Attack. Billet et al. (Billet et al., 2005) demonstrated that the white-box implementation of AES is vulnerable to a kind of algebraic attack. The attack is named after the initials of authors, the BGE attack. However, the BGE attack strongly relies on public constants and building blocks used in the AES and cannot be applied to the white-box implementation of KCipher-2.

Code-lifting Attack. A code-lifting attack extracts and copies the entire white-box implementation to other PCs and devices. The copied program can access the secret key as well as the original one since the key is embedded in the program. We can protect against this attack by increasing the storage size according to the concept of the *space hardness* (Bogdanov and Isobe, 2015).

Our original method uses four-bitwise operations. We can use eight-bitwise or 16-bitwise operations to achieve a higher level of security against a code-lifting attack. The total storage size is 3.09 gigabytes with eight-bitwise operations and over 24 petabytes with 16-bitwise operations. Thus, the implementation with eight-bitwise operations is feasible for *space hard* implementation. There exists an oblivious trade-off between portability and security against a code-lifting attack.

6 CONCLUSION

We proposed a white-box implementation of a stream cipher that can achieve the same asymptotic performance as the standard implementation. Our black-box implementation of KCipher-2 achieves low storage consumption of no more than 2 megabytes and is suitable for a PC, tablet, and smartphone. On the other hand, we can achieve *space hard* implementation to protect against a code-lifting attack. Furthermore, the implementation can protect against black-box attacks and a BGE attack. Our future research includes a performance evaluation of the software implementation and expansion of the proposed method to other stream ciphers.

REFERENCES

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (Im)possibility of Obfuscating Programs. In *Proc. of Advances in Cryptology (CRYPTO 2001), Lecture Notes in Computer Science 2139*, pages 1–18.

Billet, O., Gilbert, H., and Ech-chatbi, C. (2005). Cryptanalysis of a White Box AES Implementation. In *Proc. of Selected Areas in Cryptography (SAC 2014), Lecture Notes in Computer Science 3357*, pages 227–240.

Biryukov, A., Bouillaguet, C., and Khovratovich, D. (2014). Cryptographic schemes based on the ASASA structure: Black-box, white-box, and public-key. In *Advances in Cryptology (ASIACRYPT 2014), Lecture Notes in Computer Science 8873*, pages 63–84.

Bogdanov, A. and Isobe, T. (2015). White-Box Cryptography Revisited: Space-Hard Ciphers. In *Proc. of ACM Conference on Computer and Communications Security (ACM CCS 2015)*, pages 1058–1069.

Bos, J. W., Hubain, C., Michiels, W., and Teuwen, P. (2015). Differential Computation Analysis: Hiding your White-Box Designs is Not Enough. https://eprint.iacr.org/2015/753.

Bringer, J., Chabanne, H., and Dottax, E. (2006a). Perturbing and Protecting a Traceable Block Cipher. In *Proc. of IFIP Open Conference on Communications and Multimedia Security, Lecture Notes in Computer Science 4237*, pages 109–119.

Bringer, J., Chabanne, H., and Dottax, E. (2006b). White Box Cryptography: Another Attempt.

Cho, J., Choi, K. Y., and Moon, D. (2016). Hybrid WBC : Secure and efficient encryption schemes using the White-Box Cryptography. https://eprint.iacr.org/2016/679.

Chow, S., Eisen, P., Johnson, H., and Van Oorschot, P. C. (2003a). A White-Box DES Implementation for DRM Applications. In *Proc. of ACM Workshop on Digital Rights Management (DRM 2002), Lecture Notes in Computer Science 2696*, pages 1–15.

Chow, S., Eisen, P. a., Johnson, H., Van Oorschot, P. C., and Oorschot, P. C. V. (2003b). White-box Cryptogra-

phy and an AES Implementation. In *Proc. of Selected Areas in Cryptography (SAC 2002), Lecture Notes in Computer Science 2595*, pages 250–270.

Collberg, C., Thomborson, C., and Low, D. (1997). A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science University of Auckland.

Daemen, J. and Rijmen, V. (2002). *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer.

Fouque, P.-a., Karpman, P., Kirchner, P., and Minaud, B. (2016). Efficient and Provable White-Box Primitives. https://eprint.iacr.org/2016/642.

Goldwasser, S. and Kalai, Y. T. (2005). On the Impossibility of Obfuscation with Auxiliary. In *Proc. of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS2005)*, pages 553–562.

Goldwasser, S. and Rothblum, G. N. (2007). On Best-Possible Obfuscation. In *Proc. of Fourth IACR Theory of Cryptography Conference (TCC2007), Lecture Notes in Computer Science 4392*, pages 194–213.

Hada, S. and Sakurai, K. (2007). A Note on the (Im)possibility of Using Obfuscators to Transform Private-Key Encryption into Public-Key Encryption. In *Proc. of International Workshop on Security (IWSEC2007), Lecture Notes in Computer Science 4752*, pages 1–12.

Hofheinz, D., Malone-Lee, J., and Stam, M. (2007). Obfuscation for Cryptographic Purposes. In *Proc. of Fourth IACR Theory of Cryptography Conference (TCC2007), Lecture Notes in Computer Science 4392*, pages 214–232.

Karroumi, M. (2010). Protecting White-Box AES with Dual Ciphers. In *Proc. of Information Security and Cryptology (ICISC 2010), Lecture Notes in Computer Science 6829*, pages 278–291.

Kiyomoto, S., Tanaka, T., and Sakurai, K. (2007). K2: A Stream Cipher Algorithm Using Dynamic Feedback Control. In *Proc. of Secrypt 2007*, pages 204–213.

Klinec, D. (2013). *White-box attack resistant cryptography*. PhD thesis, Faculty of Informatics, Masaryk University.

Kocher, P. C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of Advances in Cryptology (CRYPTO 1996), Lecture Notes in Computer Scicence 1109*, pages 104–113.

Muir, J. a. (2013). A Tutorial on White-box AES. *Advances in Network Analysis and its Applications, Mathematics in Industry*, 18:209–229.

Mulder, Y. D., Roelse, P., and Preneel, B. (2013). Cryptanalysis of the XiaoLai White-Box AES Implementation. In *Proc. of Selected Areas in Cryptography (SAC 2012), Lecture Notes in Computer Science 7707*, volume Lecture No, pages 34–49.

Mulder, Y. D., Wyseur, B., and Preneel, B. (2010). Cryptanalysis of a Perturbated White-Box AES Implementation. In *Proc. of Progress in Cryptology (INDOCRYPT 2010), Lecture Notes in Computer Science 6498*, pages 292–310.

Sasdrich, P., Moradi, A., and Güneysu, T. (2016). White-Box Cryptography in the Gray Box A Hardware Implementation and its Side Channels . https://eprint.iacr.org/2016/203.

Saxena, A., Wyseur, B., and Preneel, B. (2009). Towards Security Notions for White-Box Cryptography. In *Proc. of Information Security Conference (ISC 2009), Lecture Notes in Computer Science 5735*, pages 49–58.

Xiao, Y. and Lai, X. (2009). A Secure Implementation of White-Box AES. In *Proc. of International Conference on Computer Science and its Applications (CSA 2009)*, pages 1–6.