IEICE TRANSACTIONS

on Information and Systems

DOI:10.1587/transinf.2024EDL8005

Publicized:2024/06/26

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

# CLEAR & RETURN: Stopping Run-time Countermeasures in Cryptographic Primitives

**Myung-Hyun KIM**[†], ***Nonmember and*** **Seungkwang LEE**[†a)], ***Member***

**SUMMARY** White-box cryptographic implementations often use masking and shuffling as countermeasures against key extraction attacks. To counter these defenses, higher-order Differential Computation Analysis (HO-DCA) and its variants have been developed. These methods aim to breach these countermeasures without needing reverse engineering. However, these non-invasive attacks are expensive and can be thwarted by updating the masking and shuffling techniques. This paper introduces a simple binary injection attack, aptly named *clear & return*, designed to bypass advanced masking and shuffling defenses employed in white-box cryptography. The attack involves injecting a small amount of assembly code, which effectively disables run-time random sources. This loss of randomness exposes the unprotected lookup value within white-box implementations, making them vulnerable to simple statistical analysis. In experiments targeting open-source white-box cryptographic implementations, the attack strategy of hijacking entries in the Global Offset Table (GOT) or function calls shows effectiveness in circumventing run-time countermeasures.

***key words:*** *White-box cryptography, Masking, Shuffling, Binary injection attack.*

## 1. Introduction

The primary goal of white-box cryptography is to protect secret keys from invasive attacks, particularly in environments where adversaries can fully control the software implementation. In 2002, Chow *et al.* introduced white-box implementations of AES and DES [1,2]. Their security was accomplished by converting the full-round operations into key-specific lookup tables, incorporating secret linear and nonlinear transformations. Because of the encoded lookup tables, it has become challenging for an adversary to extract the secret key solely by observing intermediate values in memory. Despite these advancements, these white-box implementations were later found to be susceptible to cryptanalysis attacks, as demonstrated by Billet *et al* [3]. An alternative attack method, Differential Computation Analysis (DCA) [4], has been developed to exploit vulnerabilities in white-box cryptography. It employs statistical analysis of computational traces to deduce the secret key, eliminating the need for reverse engineering. Contrasting with the noise-impacted power traces in Correlation Power Analysis (CPA) [5], computational traces are extracted directly from memory. They offer noise-free data on the read and write operations during encryption, significantly enhancing the precision of both analysis and key recovery.

To counteract DCA, various countermeasures have been implemented, notably run-time masking and shuffling techniques derived from the gray-box model. Masking, as a strategy, splits sensitive variables into several shares. This division and subsequent secure processing aim to mitigate the risk of information leakage [6]. Shuffling is another technique used to counteract DCA. It randomizes the sequence of independent operations, thereby disrupting the alignment of computational traces. In addition, dummy shuffling introduces non-essential operations. These dummy operations are integrated to obscure the real, sensitive computations within a redundant activities, as discussed in [7].

However, these obfuscation methods have been found vulnerable to advanced forms of DCA, such as higher-order DCA (HO-DCA) and even more sophisticated variants like higher-degree HO-DCA (HDHO-DCA) [8,9]. Despite these vulnerabilities, it is important to note that the effectiveness of higher-order attacks, like HO-DCA and HDHO-DCA, diminishes with increasing complexity. As the dimensionality of the computational traces grows, the time complexity for these higher-order attacks escalates significantly. Consequently, new implementations of masked and shuffled cryptographic systems are likely to be more resistant to these types of non-invasive attacks.

This study underscores that run-time masking and shuffling techniques in white-box cryptography are heavily reliant on random sources, such as random number generators and deterministic cryptographic algorithms. This reliance potentially exposes them to invasive attacks. For white-box cryptography to be truly effective, it needs to prove its robustness against white-box attacks. Aligned with this perspective, this paper introduces straightforward binary injection attacks, aptly named *clear & return*[†], aimed at these existing countermeasures that depend on run-time random sources. The demonstrations reveal the vulnerabilities of these countermeasures when faced with white-box attackers. To the best of our knowledge, it provides the first successful demonstration of a white-box attack targeting open-source white-box cryptographic implementations that employ masking and shuffling techniques.

---

[†] Myung-Hyun Kim and Seungkwang Lee (correponding author) are with Department of Cyber Security, Dankook University, South Korea.

a) E-mail: sk.cryptographic@dankook.ac.kr

[†] An earlier version of this study was presented as a poster at ACNS 2023, Kyoto.

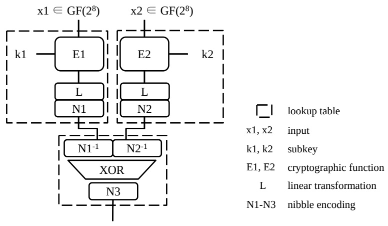


Fig. 1: Fundamental design of typical white-box cryptographic implementations.

## 2. Preliminaries

### 2.1 Fundamentals of White-box Cryptography

Typically, a white-box cryptographic approach involves a network of obfuscated lookup tables. These tables collectively perform the operations of a cryptographic algorithm in a concealed manner. Consider an illustrative example: for a cryptographic operation $y = E_k(x)$, with $y$, $x$, $k$ belonging to GF($2^8$) and $k$ representing a subkey, one can conceptualize $\mathcal{T}_k$, an $8\times 8$ table, executing the mapping from $x$ to $y$. This table, $\mathcal{T}$, is then obscured with secret and invertible transformations to shield the key from extraction by white-box adversaries, based on the observed inputs and outputs. These transformations, denoted as $G$ and $F$, lead to the equation: $\mathcal{T}_k = G \circ E_k \circ F^{-1}$. Note that these transformations include both linear and nonlinear components.

Fig. 1 demonstrates a key architectural element in fundamental white-box cryptographic schemes, specifically applied to a basic cryptographic operation denoted as $E1(x1, k1) \oplus E2(x2, k2)$. This design incorporates an encoding stage that consists of a linear transformation, followed by a sequence of two concatenated nibble encodings. It employs a consistent pair of matrices for linear transformations in adjacent lookup tables. By using this method, the process of creating a lookup table for the XOR operation is simplified, as it eliminates the need to decode the linear transformation. This is fundamentally reliant on the distributive property of multiplication over addition.

### 2.2 Techniques of Masking and Shuffling

To counteract the correlation issues in side-channel attacks, software-based strategies like masking and operation shuffling are widely employed. Masking involves dividing a sensitive variable $s$ into $d+1$ segments, labeled $s_0, s_1, \ldots, s_d$. These segments are combined using a group operation $\circ$ (commonly XOR $\oplus$ or modular addition) to reconstruct the original variable $s$. Crucially, any subset of these segments, fewer than $d+1$, or their leakage signals, should be statistically unrelated to $s$. This is ensured by randomly selecting the masks $s_1, \ldots, s_d$, and calculating the primary masked segment $s_0$ as $s_0 = s \circ s_1 \circ \ldots \circ s_d$. The term 'masking order' refers to the value of $d$.

In addition to masking, operation shuffling is another technique employed to enhance security in cryptographic systems. Shuffling involves rearranging the order of operations or the execution sequence of a cryptographic algorithm. By randomly permuting these operations, shuffling makes it more challenging for an attacker to predict or analyze the data flow and intermediate states of the algorithm, thus further obfuscating side-channel signals. This randomized rearrangement is particularly effective in mitigating time-based side-channel attacks, as it disrupts the consistent timing patterns that such attacks often rely on.

## 3. Proposed Method

### 3.1 Key Idea Behind and Assumption

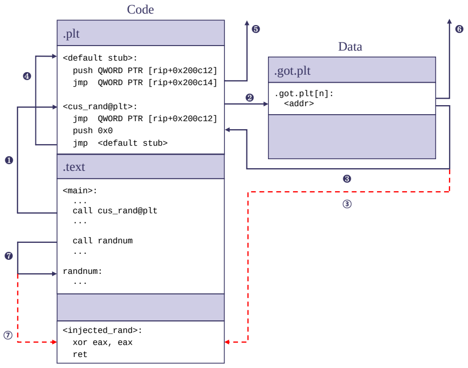
To defeat aforementioned countermeasures, run-time random sources must be disabled so that key-dependant intermediate values are exploitable. Generally speaking, a random number generator can be implemented by using either shared libraries or user-defined functions. In this section, we introduce simple binary injection attacks, enforcing the random number generator to always output a fixed value.

One of the easy ways to produce a sequence of random numbers is to call a function, such as `rand()`, provided in shared libraries. Lazy binding in Linux ELF binaries resolves unknown references to functions located in shared libraries, using the *Procedure Linkage Table* (.plt) and the *Global Offset Table* (.got) sections. In other words, the address of `rand()` can be found in its GOT entry once it is called. The steps ❶ - ❻ shown in Fig. 2 present the overall procedure of lazy binding to call `rand()` for the first time.

In Unix-like systems, overwriting the GOT entries is a traditional control flow hijacking technique. Leveraging the dynamic symbol binding mechanism, this attack modifies a GOT entry to redirect the program's execution flow to a chosen target address. Previously, it was commonly employed to tamper with the GOT entry of a shared library function, often substituting it with `system()`, thereby enabling the execution of `/bin/sh` and spawning a shell [10]. In our attack scenario, by substituting the GOT entry with the address of injected code, the attacker can manipulate the behavior of the `rand()` function within the context of side-channel analysis. Essentially, if the injected code consistently returns zero, the protection of sensitive variables is compromised.

Utilizing user-defined functions or static libraries for random number generation is independent of the GOT entry, rendering GOT entry hijacking ineffective. In such scenarios, an attacker would need to identify and modify the specific function calls, replacing them with manipulated calls to their injected code. It is noteworthy that numerous established countermeasures leverage cryptographic functions to produce a sequence of uniformly distributed random numbers. Standard block ciphers are frequently employed in these countermeasures to ensure a high entropy in the generated sequence.

In the following, we present *clear & return*, a binary in-



```
<default stub>:
  push QWORD PTR [rip+0x200c12]
  jmp  QWORD PTR [rip+0x200c14]

<cus_rand@plt>:
  jmp  QWORD PTR [rip+0x200c12]
  push 0x0
  jmp  <default stub>
```

```
.got.plt[n]:
  <addr>
```

```
<main>:
  ...
  call cus_rand@plt
  ...

  call randnum
  ...

randnum:
  ...
```

```
<injected_rand>:
  xor eax, eax
  ret
```

Fig. 2: Overview of the proposed attack. Solid line: benign function calls, Dashed line: hijacked function calls.

jection attack on open-source implementations of white-box cryptography that have applied masking and shuffling techniques to enhance security. It is assumed that the compiled binaries operate on Intel x86 CPUs and adhere to the System V-style calling convention, where function return values are stored in the EAX (or RAX) register. Also, it is assumed that the symbols in the binaries are not stripped. Based on these assumptions, we downloaded the source codes and built their binaries based on shared libraries. The process of neutralizing the random number generator and extracting the secret key from unprotected intermediates will be detailed in the following section.

### 3.2 *Clear & Return*: Disabling Random Sources

To introduce our attack *clear & return*, we utilized two open-source white-box AES implementations. The first one is based on masking [†], while the second applies shuffling techniques [7, 11][††]. We will refer to these implementations as Mask-WB-AES and Shuff-WB-AES, respectively.

Mask-WB-AES mainly relies on a library function with the signature `uint8_t cus_rand()` while Shuf-WB-AES depends on a user-defined function `uint8_t randnum()` to generate random numbers. To provide a comprehensive understanding, we have consolidated two individual attacks against these implementations into a unified illustration, as depicted in Fig. 2.

Our injected code effectively undermines traditional countermeasures such as masking and shuffling employed in white-box cryptography by compromising the sources of randomness. As illustrated in Fig. 2, step ③ demonstrates the hijacking of the GOT entry for the `cus_rand()` function. Normally, `cus_rand()` generates an integer in the range of 0 to 255. However, our manipulation ensures that it invariably returns zero. We refer to this tactic as *clear & return*, and it is evidenced by the code segment marked as `<injected_rand>`. Similarly, step ⑦ illustrates our method of intercepting the `randnum()` function calls, further emphasizing the comprehensive nature of our attack on stopping randomness capabilities. The return values from these functions, which are conventionally stored in the EAX register, are thus overridden to neutralize the intended cryptographic protection.

It is important to note that if a user-defined function transmits the address of the random number via an argument, a different register must be filled with a reference to zeros. The specific register to be manipulated is determined based on the argument's position, in accordance with the System V-style calling convention.

### 3.3 Experimental Results

To evaluate our attacks' effectiveness, we encrypted 1,000 random plaintexts for both Mask-WB-AES and Shuf-WB-AES. To collect computational traces, we employed a dynamic binary instrumentation (DBI) tool using Valgrind, capturing data from memory write operations during the encryption process. Subsequent CPA attacks were executed, with the outcomes depicted in Fig. 3. The figure illustrates the success of our binary injection in halting random number generators. This interruption effectively neutralizes masking and shuffling defenses, enabling the extraction of correct subkeys from a mere 1,000 computational traces.

An additional observation is that the target open-source code may not properly apply internal encoding to protect the intermediate values of the white-box cryptographic implementation. This possibility is suggested by the fact that the hypothetical intermediate value computed for the correct subkey results in a Pearson correlation coefficient of 1. Generally, in CPA attacks against white-box cryptographic implementations with correctly implemented internal encoding, the correlation coefficient does not typically reach the maximum value of 1.

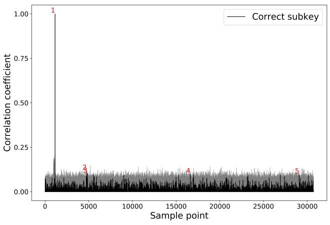
### 3.4 Discussion

Several protection techniques exist to prevent memory corruption [12]. For instance, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) primarily aim to secure the stack or heap. Additionally, the address layout of code and data sections can be randomized through Position Independent Executable (PIE). To defend against the GOT overwriting attacks, RELocation Read Only (Relro) serves as a barrier against runtime tampering by establishing the data section, utilized by the dynamic linker, as read-only during program loading. In particular, Full Relro extends protection to the GOT section as well. In a program protected by Full Relro, the GOT becomes read-only after all library function calls are bound during loading, preventing runtime GOT modification. Notably, the proposed attack persists unimpeded by these countermeasures due to two key factors. Firstly, the injected section's placement falls beyond the original address space, rendering it impervious to the randomization introduced by these defenses. Secondly, our

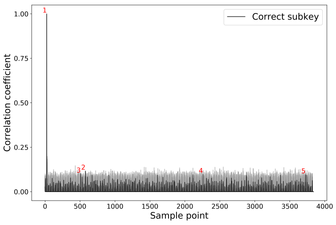
---

[†] github.com/Nexus-TYF/BU-White-box-AES/tree/main

[††] github.com/scnucrypto/HO-DCA/tree/main /a_specific_BU_shuffling



(a) CPA result on Mask-WB-AES



(b) CPA result on Shuf-WB-AES

Fig. 3: CPA results after stopping random number generator. Correct subkey in black, wrong subkey candidates in gray, rank of correct subkey highlighted in red.

code injection and GOT overwriting take place before the binary is loaded, rendering run-time protection ineffective.

Indeed, our binary injection could be thwarted through various anti-tampering techniques, including obfuscation, integrity checks, and binary encryption. Among these methods, integrity checks on the binary stand out as the most straightforward and widely applicable approach. However, implementing such verification processes often involves resource-intensive cryptographic operations like hashing and signature algorithms. Consequently, cryptographic functions in low-cost devices such as IoT devices may still remain vulnerable to our attack, as these devices are typically not protected by integrity checks.

## 4. Conclusion & Future Work

This paper demonstrated the binary injection attack, capable of defeating the cryptographic countermeasures that are dependent on run-time random sources. We redirected the GOT entries and the calls to user-defined functions to our injected code, consistently producing zeros instead of random numbers. To protect the target binary from a binary injection attack, various binary anti-tampering techniques can be employed. For instance, integrity checks including hash-based checks, checksums, and digital signatures can be utilized to detect modifications to the binary code. Our future work includes several key initiatives. First, we aim to conduct a performance comparison detailing the number of computational traces necessary to recover a subkey and the time elapsed between HO-DCA, HDHO-DCA, and DCA using our injected code. Additionally, we plan to demonstrate the hijacking of calls to various cryptographic functions, each with distinct prototypes. Last but not least, we intend to develop an automation tool capable of detecting all random sources in binaries and customizing code injections based on the characteristics of each random source.

## Acknowledgement

The present research was supported by the research fund of Dankook University in 2022. Special thanks to Prof. Taek-Young Youn for his valuable support throughout the development of this work.

### References

[1] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, “A white-box des implementation for drm applications,” Digital Rights Management, ed. J. Feigenbaum, Berlin, Heidelberg, pp.1–15, Springer Berlin Heidelberg, 2003.

[2] S. Chow, P.A. Eisen, H. Johnson, and P.C.v. Oorschot, “White-box cryptography and an aes implementation,” Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography, SAC ’02, Berlin, Heidelberg, p.250–270, Springer-Verlag, 2002.

[3] O. Billet, H. Gilbert, and C. Ech-Chatbi, “Cryptanalysis of a white box aes implementation,” Selected Areas in Cryptography, ed. H. Handschuh and M.A. Hasan, Berlin, Heidelberg, pp.227–240, Springer Berlin Heidelberg, 2005.

[4] J.W. Bos, C. Hubain, W. Michiels, and P. Teuwen, “Differential computation analysis: Hiding your white-box designs is not enough,” Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings, ed. B. Gierlichs and A.Y. Poschmann, Lecture Notes in Computer Science, vol.9813, pp.215–236, Springer, 2016.

[5] E. Brier, C. Clavier, and F. Olivier, “Correlation Power Analysis with a Leakage Model,” Proceedings of the 6th International Workshop on Cryptographic Hardware and Embedded Systems, CHES ’04, pp.16–29, Springer, 2004.

[6] O. Seker, T. Eisenbarth, and M. Liskiewicz, “A white-box masking scheme resisting computational and algebraic attacks,” IACR Transactions on Cryptographic Hardware and Embedded Systems, vol.2021, no.2, p.61–105, Feb. 2021.

[7] A. Biryukov and A. Udovenko, “Dummy shuffling against algebraic attacks in white-box implementations,” Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part II, Berlin, Heidelberg, p.219–248, Springer-Verlag, 2021.

[8] A. Bogdanov, M. Rivain, P.S. Vejre, and J. Wang, “Higher-order DCA against standard side-channel countermeasures,” Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings, ed. I. Polian and M. Stöttinger, Lecture Notes in Computer Science, vol.11421, pp.118–141, Springer, 2019.

[9] Y. Tang, Z. Gong, J. Chen, and N. Xie, “Higher-order dca attacks on white-box implementations with masking and shuffling countermeasures,” IACR Transactions on Cryptographic Hardware and Embedded Systems, vol.2023, no.1, p.369–400, Nov. 2022.

[10] c0ntex, “How to hijack the global offset table with pointers for root shells.” Accessed: 2024-04-01.

[11] A. Biryukov and A. Udovenko, “Attacks and countermeasures for white-box designs,” Advances in Cryptology – ASIACRYPT 2018, ed. T. Peyrin and S. Galbraith, Cham, pp.373–402, Springer International Publishing, 2018.

[12] M.A. Butt, Z. Ajmal, Z.I. Khan, M. Idrees, and Y. Javed, “An in-depth survey of bypassing buffer overflow mitigation techniques,” Applied Sciences, vol.12, no.13, 2022.