Integrative Acceleration of First-Order Boolean Masking for Embedded IoT Devices

Yuichi Komano¹, Hideo Shimizu¹, and Hideyuki Miyake¹

Toshiba Corporation, Kawasaki, Japan
{yuichi1.komano,hideo.shimizu,hideyuki.miyake}@toshiba.co.jp

Abstract. Physical attacks, especially side-channel attacks, are threats to IoT devices which are located everywhere in the field. For these devices, the authentic functionality is important so that the IoT system becomes correct, and securing this functionality against side-channel attacks is one of our emerging issues. Toward that, Coron et al. gave an efficient arithmetic-to-Boolean mask conversion algorithm which enables us to protect cryptographic algorithms including arithmetic operations, such as hash functions, from the attacks. Recently, Biryukov et al. improved it by locally optimizing subroutines of the conversion algorithm. In this paper, we revisit the algorithm. Unlike Biryukov et al., we improve the Coron et al.'s algorithm with integrative optimizations over the subroutines. The gains against these algorithms are about 22.6% and 7.0% in the general setting. We also apply our algorithm to HMAC-SHA-1 and have an experiment to show that the implementation on a test vehicle smartcard leaks no sensitive information with the ISO/IEC17825 test.

Keywords: side-channel attack, mask conversion, IoT, embedded device

1 Introduction

Internet of Things (IoT) has been widely spread to make our lives smart and comfortable. In the IoT system, devices are located in the field and communicate each other to collect their sensing data and to control actuators.

Securing the devices in the IoT system is one of emerging issues. Especially, side-channel attacks including the power analyses [10,11,4] should be serious threats to the devices. This is because, in the IoT system, attackers easily get hold of devices from the field and physically analyze them. Moreover, most of cost-constrained devices lack the tamper protection mechanism and they can be vulnerable to such attacks.

Among the security functionalities of the IoT devices, the authenticity is strongly required more than others, such as the confidentiality and the anonymity, to keep the system correct. In this paper, we therefore focus on the countermeasure against the side-channel attack which realizes the secure authentication.
1.1 First-Order Boolean and Arithmetic Maskings

The hash-based message authentication code (HMAC, [2, 18]) is widely used for the authentication. Unlike the block cipher, like AES [17], each of the secure hash algorithms (SHA, [19]) includes arithmetic additions besides Boolean operations. Against the HMAC with SHA, side-channel vulnerabilities have been reported [13, 15, 1].

In order to protect the HMAC against the side-channel attack, both of the Boolean and arithmetic operations should be randomized with different types of masks. Hence, algorithms, converting a Boolean mask to an arithmetic one and vice versa, are required.

Goubin [6] proposed promising conversion algorithms for both directions. Since then, improvements and extensions, such as ones for higher-order masking, have been reported. However, to the best of our knowledge, his Boolean-to-arithmetic conversion is the best algorithm with fewer operations, up to now.

Recently, Coron et al. [5] proposed an arithmetic-to-Boolean conversion by changing the basis, an arithmetic addition, of the algorithm. Unlike the previous conversion based on the ripple-carry adder, they revisited the Kogge-Stone carry look-ahead adder [12] to construct a new conversion. With this approach, the amount of computation is dramatically decreased from $O(k)$ to $O(\log k)$ where $k$ is the addition bit size. They also gave a masked addition algorithm which computes, with inputs randomized with Boolean masks, a sum with one of the Boolean masks.

In CARDIS 2017, Biryukov et al. [3] improved the Coron et al.’s masked addition. They searched optimal subroutines called in the Coron et al.’s masked addition in formal manner and applied them to construct an improved masked addition algorithm. They also applied their masked addition algorithm to the lightweight block ciphers, SIMON, SPECK, and RECTANGLE.

In addition to the above two, Schneider et al. [20] discussed an efficient hardware implementation of the conversion. Won and Han [21] modified the Kogge-Stone carry look-ahead adder with a divide and conquer approach.

1.2 Our Contributions

In this paper, we propose improved algorithms of the Coron et al.’s conversion and masked addition algorithms. Biryukov et al. took a bottom-up approach to individually optimize subroutines; however, our approach is integrative. We decrease the number of mask operations beyond the subroutines. Our tricks are to unify the subroutines\(^1\) and to break the symmetry of subroutine calls in iterations.

We succeeded to decrease the coefficient (for $\log_2 k$ in Table 1) in the computational complexity by seven and one against the Coron et al.’s conversion and the Biryukov et al.’s one, respectively. Against the Coron et al.’s algorithm, Biryukov et al.’s one requires fewer numbers of AND and XOR operations

\(^1\) Recently, Jungk et al. [9] also proposed more efficient algorithms independently.
whereas it additionally executes the OR (or\textit{r}) operations. On the other hand, our algorithm focuses to decrease the numbers of XOR operations; and eventually, ours requires fewer operations than both algorithms in total (as shown in Table 2). The gains against the previous conversions are 22.6\% and 7.0\% if the size of addition unit is supposed to be $k = 32$.

We then applied our conversion algorithm to protect the authentication with HMAC-SHA-1. The gains, on the theoretical numbers of required operations, are still 17.1\% and 4.9\% against the HMAC-SHA-1 implementations with previous conversion algorithms (as shown in Table 3). We also developed a software implementation for an IC test vehicle smartcard [7] from Information-technology Promotion Agency (IPA), Japan. In this software, we use the xorshift [14] as a mask generation function. From the power consumption traces measured with the smartcard, the testing method of ISO/IEC 17825 [8] finds no vulnerability which confirms the security of our conversion.

1.3 Organization

The remaining of this paper is organized as follows. Section 2 reviews the previous works. In Section 3, we explain our strategy and propose our arithmetic-to-Boolean conversion and masked addition algorithms. We then check the first-order security of our algorithm with the IPA test vehicle smartcard in Section 4. Section 5 gives discussions on the detail and the extension of our algorithms. Finally, Section 6 concludes this paper.

2 Related Works

In this section, we review previous works related to the maskings.

2.1 Goubin’s Mask Conversion Algorithm [6]

We recall the Boolean-to-arithmetic conversion algorithm by [6]. From $x' = x \oplus r$ and $r$, this algorithm efficiently computes $A = x - r$ as in Algorithm 1. We use this algorithm in our experiments later. In [6], Goubin also proposed a reverse conversion, the arithmetic-to-Boolean conversion, but we omit its detail in this paper.

2.2 Coron et al.’s Algorithms [5]

Coron et al. [5] proposed an efficient arithmetic-to-Boolean conversion applicable to the first-order masking. They took a new approach to construct their conversion by securing the Kogge-Stone carry look-ahead adder [12], although the Goubin’s conversion was based on the ripple-carry adder. They also proposed a Kogge-Stone masked addition which, with inputs with Boolean masks, computes a sum with one of the Boolean masks for inputs.
Algorithm 1 Goubin’s Boolean-to-arithmetic Conversion Algorithm

Input: \( x', r \in \{0, 1\}^k \) such that \( x' = x \oplus r \) for secret \( x \in \{0, 1\}^k \)
Output: \( A \in \mathbb{F}_{2^k} \) and \( r \) such that \( A = x - r \)

1: \( \gamma \leftarrow \{0, 1\}^k \)
2: \( t \leftarrow x' \oplus \gamma \)
3: \( t \leftarrow t - \gamma \)
4: \( t \leftarrow t \oplus x' \)
5: \( \gamma = \gamma \oplus r \)
6: \( a \leftarrow x' \oplus \gamma \)
7: \( a \leftarrow a - \gamma \)
8: \( a \leftarrow a \oplus t \)
9: \text{return} \( a, r \)

In [5], they also reported the implementations of HMAC-SHA-1 with first-order masking with the Kogge-Stone arithmetic-to-Boolean conversion and the Kogge-Stone masked addition. From their result, the implementation with their masked addition is less effective, requiring about 2.28 times clock cycles, compared to one with their conversion. Hence, this section only review the Kogge-Stone arithmetic-to-Boolean mask conversion as depicted in Algorithm 2.

Algorithm 2 calls subroutines [5] labeled as SecShift, SecAnd, and SecXor which securely execute operations of Shift, AND, and XOR, respectively, by using masks.

2.3 Biryukov et al.’s Masked Addition Algorithm [3]

Biryukov et al. [3] took a comprehensive approach to search optimal algorithms of subroutines which securely execute operations of AND and OR. They then applied these subroutines to construct improved masked addition and subtraction algorithms.

Algorithm 3 shows the improved masked addition algorithm from [3]. This algorithm calls the improved SecAnd labeled as SecAnd2, and two other subroutines labeled as SecShift2 and SecXor2.

3 New Algorithms

We first explain our strategy to improve the arithmetic-to-Boolean conversion algorithm and the masked addition. We then give our algorithms and compare their efficiencies with ones of the previous algorithms.

3.1 Strategy

Our approach is to improve the Coron et al.’s algorithm. We have two ideas to enhance this algorithm.

The first one is to decrease the numbers of required masks from three to two, by replacing the third mask (\( t \) in Algorithm 2) with an XOR of other two masks.

Input: $A, r \in \{0, 1\}^k$ and $n = \max(\lceil \log_2(k - 1) \rceil, 1)$ such that $A = x - r \in \mathbb{F}_{2^n}$

Output: $x'$ such that $x' \equiv r \mod 2^k$

1: Let $s \leftarrow \{0, 1\}^k, t \leftarrow \{0, 1\}^k, u \leftarrow \{0, 1\}^k$
2: $P' \leftarrow A \oplus s$
3: $P' \leftarrow P' \oplus r$
4: $G' \leftarrow s \oplus ((A \oplus t) \land r)$
5: $G' \leftarrow G' \oplus (t \land r)$
6: for $i := 1$ to $n - 1$ do
7:     $H \leftarrow \text{SecShift}(G', s, t, 2^{i-1})$
8:     $U \leftarrow \text{SecAnd}(P', H, s, t, u)$
9:     $G' \leftarrow \text{SecXor}(G', U, u)$
10:    $H \leftarrow \text{SecShift}(P', s, t, 2^{i-1})$
11:   $P' \leftarrow \text{SecAnd}(P', H, s, t, u)$
12:  $P' \leftarrow P' \oplus s$
13:  $P' \leftarrow P' \oplus u$
14: end for
15: $H \leftarrow \text{SecShift}(G', s, t, 2^{n-1})$
16: $U \leftarrow \text{SecAnd}(P', H, s, t, u)$
17: $G' \leftarrow \text{SecXor}(G', U, u)$
18: $x' \leftarrow A \oplus 2G'$
19: $x' \leftarrow x' \oplus 2s$
20: return $x'$

(as in the second line in Algorithm 5). This replacement degrades the level of higher-order security by one; however, masking an internal variable with one of two independent masks and their XOR value is enough to ensure the first-order security. This replacement decreases not only the number of masks itself but also that of XOR operations (for Steps 12 and 13 in Algorithm 2, etc.).

The second one is to decrease the number of re-masking operations within subroutines. In their algorithm, SecShift is followed by SecAnd. In each of SecShift and SecAnd, XOR operations are executed in order for the output to be masked with a certain mask. Our second idea is to remove the operations for re-masking by integrating these two subroutines (named SecShiftAnd in Algorithm 4).

In addition to above two, we also change the initialization steps (Steps 3 to 5 in Algorithm 2) to decrease the number of XOR operations (with input $A$).

Although our ideas seem to naturally lead an improved algorithm, unfortunately, it is incorrect. This is because, by reducing the number of independent masks, we have to use the mask in different order from the original algorithm so that masks are uncanceled. As seen in the next subsection, we prepare two sequences of operations (Steps 8 to 11 and 13 to 15, etc.) to keep the internal variables being masked throughout the conversion.
**Algorithm 3** Biryukov et al.'s Masked Addition Algorithm [3]

**Input:** $x_1, x_2, y_1, y_2 \in \{0, 1\}^k$ and $n = \max(\lfloor \log_2(k-1) \rfloor, 1)$ such that $x = x_1 \oplus x_2$ and $y = y_1 \oplus y_2$

**Output:** $z_1, z_2$ such that $z = z_1 \oplus z_2 = (x + y) \mod 2^k$

1. $p_1, p_2 \leftarrow \text{SecXor}2(x_1, x_2, y_1, y_2)$
2. $g_1, g_2 \leftarrow \text{SecAnd2}(x_1, x_2, y_1, y_2)$
3. $g_1, g_2 \leftarrow ((g_1 \oplus x_2) \oplus g_2, x_2)$
4. for $i := 1$ to $n - 1$ do
   5. $h_1, h_2 \leftarrow \text{SecShift2}(g_1, g_2, 2^{i-1})$
   6. $u_1, u_2 \leftarrow \text{SecAnd2}(p_1, p_2, h_1, h_2)$
   7. $g_1, g_2 \leftarrow \text{SecXor2}(g_1, g_2, u_1, u_2)$
   8. $h_1, h_2 \leftarrow \text{SecShift2}(p_1, p_2, 2^{n-1})$
   9. $h_1, h_2 \leftarrow ((h_1 \oplus x_2) \oplus h_2, x_2)$
10. $p_1, p_2 \leftarrow \text{SecAnd2}(p_1, p_2, h_1, h_2)$
11. $g_1, g_2 \leftarrow \text{SecXor2}(g_1, g_2, u_1, u_2)$
12. end for
13. $h_1, h_2 \leftarrow \text{SecShift2}(g_1, g_2, 2^{n-1})$
14. $u_1, u_2 \leftarrow \text{SecAnd2}(p_1, p_2, h_1, h_2)$
15. $g_1, g_2 \leftarrow \text{SecXor2}(g_1, g_2, u_1, u_2)$
16. $z_1, z_2 \leftarrow \text{SecXor2}(y_1, y_2, x_1, x_2)$
17. $z_1, z_2 \leftarrow (z_1 \oplus 2g_1, z_2 \oplus 2g_2)$
18. return $z_1, z_2$

**Algorithm 4** SecShiftAnd

**Input:** $x_i', s_i, j, x_2, s_2, u$ such that $x_i', s_i, u \in \{0, 1\}^k$ and $j \in \mathbb{Z}$ where $x_i = x_i \oplus s_i$

**Output:** $z'$ such that $z' = ((x_1 \ll j) \land x_2) \oplus u$

1. $y \leftarrow x_i' \ll j$
2. $s' \leftarrow s_i \ll j$
3. $z' \leftarrow u \oplus (x_2' \land y)$
4. $z' \leftarrow z' \oplus (x_2' \land s')$
5. $z' \leftarrow z' \oplus (s_2 \land y)$
6. $z' \leftarrow z' \oplus (s_2 \land s')$
7. return $z'$

3.2 Algorithms

As mentioned, we introduce a combined subroutine SecShiftAnd to accelerate the conversion. Algorithm 4 gives its procedure, which is naturally derived by combining SecShift and SecAnd of [5] without the XOR operations in SecShift.

We then give our conversion in Algorithm 5. As mentioned in the previous subsection, we use two sequences of operations to be selectively used by the conditions of $i$ (loop counter) and $n$. Note that these conditions are public and the branches leak no sensitive information of the internal variables.
Algorithm 5: Our Arithmetic-to-Boolean Conversion

Input: \( A, r \in \{0, 1\}^k \) and \( n = \max(\lceil \log_2(k - 1) \rceil, 1) \) such that \( A = x - r \in \mathbb{F}_{2^k} \)

Output: \( x' \) such that \( x' + r = A + r \text{ mod } 2^k \)

1: \( s, u \leftarrow \{0, 1\}^k \)
2: \( t \leftarrow s \oplus u \)
3: \( P' \leftarrow A \oplus s \)
4: \( G' \leftarrow t \oplus (P' \land r) \)
5: \( G' \leftarrow G' \oplus (s \land r) \)
6: \( P' \leftarrow P' \oplus r \)
7: \text{for } i := 1 \text{ to } n - 1 \text{ do}
8: \quad \text{if } i \text{ is odd then}
9: \quad \quad U \leftarrow \text{SecShiftAnd}(G', t, 2^{i-1}, P', s, u) \)
10: \quad \quad G' \leftarrow G' \oplus U \)
11: \quad \quad P' \leftarrow \text{SecShiftAnd}(P', s, 2^{i-1}, P', s, t) \)
12: \quad \quad \quad \text{else}
13: \quad \quad \quad U \leftarrow \text{SecShiftAnd}(G', s, 2^{i-1}, P', t, u) \)
14: \quad \quad \quad G' \leftarrow G' \oplus U \)
15: \quad \quad \quad P' \leftarrow \text{SecShiftAnd}(P', t, 2^{i-1}, P', t, s) \)
16: \quad \quad \text{end if}
17: \quad \text{end for}
18: \quad \text{if } n \text{ is odd then}
19: \quad \quad U \leftarrow \text{SecShiftAnd}(G', t, 2^{n-1}, P', s, u) \)
20: \quad \text{else}
21: \quad \quad U \leftarrow \text{SecShiftAnd}(G', s, 2^{n-1}, P', t, u) \)
22: \quad \text{end if}
23: \quad G' \leftarrow G' \oplus U \)
24: \quad x' \leftarrow A \oplus 2G' \)
25: \quad \text{if } n \text{ is odd then}
26: \quad \quad x' \leftarrow x' \oplus 2s \)
27: \quad \text{else}
28: \quad \quad x' \leftarrow x' \oplus 2t \)
29: \quad \text{end if}
30: \text{return } x'

3.3 Comparison with Previous Algorithms

Let us compare our algorithm with the previous ones. Table 1 summarizes the numbers of operations required in each algorithm. In [3], Biryukov et al. gave, not an arithmetic-to-Boolean algorithm, but the masked addition and subtraction algorithms. From their masked addition algorithm, an arithmetic-to-Boolean conversion algorithm are naturally derived, which we call the Biryukov et al.’s conversion algorithm. We give its detail as Algorithm 7 in Appendix A.

This table shows that our conversion and addition algorithms require fewer cycles compared to previous ones. For example, the gains of our conversion algorithm against the Coron et al.’s and Biryukov et al.’s ones, for \( k = 32 \), are 31 (\( \approx 22.6\% \)) and 8 (\( \approx 7.0\% \)), respectively.
Algorithm 6 Our Masked Addition Algorithm

Input: $x', y', r, s \in \{0,1\}^k$ and $n = \max(\lceil \log_2(k - 1) \rceil, 1)$ such that $x' = x \oplus r$ and $y' = y \oplus s$

Output: $z'$ such that $z \oplus r = x + y \mod 2^k$

1: $u \leftarrow \{0,1\}^k$
2: $t \leftarrow s \oplus u$
3: $z' \leftarrow z' \oplus y'$
4: $P' \leftarrow z' \oplus r$
5: $z' \leftarrow z' \oplus s'$
6: $G' \leftarrow \text{SecAnd}(x', y', s, r, t)$
7: for $i := 1$ to $n - 1$ do
8:     if $i$ is even then
9:         $U \leftarrow \text{SecShiftAnd}(G', t, 2^{i-1}, P', s, u)$
10:        $G' \leftarrow G' \oplus U$
11:        $P' \leftarrow \text{SecShiftAnd}(P', 2^{i-1}, P', s, u)$
12:     else
13:         $U \leftarrow \text{SecShiftAnd}(G', s, 2^{i-1}, P', t, u)$
14:        $G' \leftarrow G' \oplus U$
15:        $P' \leftarrow \text{SecShiftAnd}(P', 2^{i-1}, P', t, s)$
16:     end if
17: end for
18: if $n$ is even then
19:    $U \leftarrow \text{SecShiftAnd}(G', t, 2^{n-1}, P', s, u)$
20: else
21:    $U \leftarrow \text{SecShiftAnd}(G', s, 2^{n-1}, P', t, u)$
22: end if
23: $G' \leftarrow G' \oplus U$
24: $z' \leftarrow z' \oplus 2G'$
25: if $n$ is even then
26:    $z' \leftarrow z' \oplus 2s$
27: else
28:    $z' \leftarrow z' \oplus 2t$
29: end if
30: return $z'$

Let us check the detail of the conversion algorithm. Table 2 summarizes the number of operations required in each algorithm for $k = 32$. Compared to the Coron et al.’s conversion algorithm, although the Biryukov et al.’s one additionally requires $20$ or $n$ operations, it decreases the numbers for and and eor operations. In total, the gain is $23$.

Our conversion algorithm, on the other hand, decreases the number of operations (only) for eor by $31$ from the Coron et al.’s one. The gain is enough large to compensate the overhead on and operation against the Biryukov et al.’s one.
Algorithm rand $k = 8$ $k = 16$ $k = 32$ $k = 64$ $k$
Coron et al.'s conversion 3 81 109 137 165 $28 \log_2 k - 3$
Biryukov et al.'s conversion 2 70 92 114 136 $22 \log_2 k + 4$
Our conversion 2 64 85 106 127 $21 \log_2 k + 1$
Coron et al.'s addition 2 88 116 144 172 $28 \log_2 k + 4$
Biryukov et al.'s addition 0 70 92 114 136 $22 \log_2 k + 4$
Our addition 1 69 90 111 132 $21 \log_2 k + 6$

Table 1. Number of operations in each algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>and</th>
<th>orn</th>
<th>sft</th>
<th>xor</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coron et al.'s conversion</td>
<td>38</td>
<td>0</td>
<td>20</td>
<td>79</td>
<td>137</td>
</tr>
<tr>
<td>Biryukov et al.'s conversion</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>54</td>
<td>114</td>
</tr>
<tr>
<td>Our conversion</td>
<td>38</td>
<td>0</td>
<td>20</td>
<td>48</td>
<td>106</td>
</tr>
</tbody>
</table>

Table 2. Number of operations in each algorithm for $k = 32$ (in detail)

4 Experiments

We apply our conversion algorithm to give a first-order secure implementation of HMAC-SHA-1. In considering the Coron et al.'s result, we implement it, not with the masked addition, but with Goubin’s Boolean-to-arithmetic conversion algorithm and ours. Namely, internal values of (HMAC-)SHA-1 is basically randomized with a first order Boolean mask. In each round operation, we use the rapid Goubin’s algorithm to convert inputs with arithmetic masks suitable for additions, and our algorithm once to convert the summation back to the data with a Boolean mask.

4.1 Equipments

We developed a software for HMAC-SHA-1 [2, 18, 19] with MDK-lite for Windows, version 5.24.1. The C code was compiled with armcc v5.06 update 5 (build 528) using the O3 optimization to generate an assembly code. We then checked the assembly code not to remove the masking and reverted the code to retrieve the masking back in the assemble level. Finally, the assembly code was compiled with armasm v5.06 update 5 (build 528) to generate a binary code.

We then downloaded the binary code into an IC test vehicle smartcard [7] from Information-technology Promotion Agency (IPA), Japan. This smartcard includes the ARM7 based SC100 with 28MHz system clock, the 512KB flash memory and the 18KB RAM. The interface follows the ISO7816-3 with T=0.

We measured power consumption traces from the smartcard with the SASEBO-W board [16] and the digital oscilloscope LECROY WavePro715Zi. We controlled the SASEBO-W board, with external 2.5V power supply and 3.57MHz
frequency, from a Windows based laptop PC to run the smartcard. We acquired power consumption traces using the oscilloscope with 1G Samples/s.

4.2 Implementation of HMAC-SHA-1

Prototype with Python: Before implementing with C, we first implemented HMAC-SHA-1 with Python from scratch. Table 3 summarizes the number of randomnsses and operations required for HMAC-SHA-1 with each algorithm. For the latter, we count the operations of \texttt{add}, \texttt{sub}, \texttt{and}, \texttt{or}, \texttt{eor}, \texttt{orr} (only for the Biryukov et al.’s conversion), \texttt{shift}, and \texttt{rot} which are supposed to be executed in one clock cycle with ARM processor.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>#rand</th>
<th>#ops</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without countermeasure</td>
<td>0</td>
<td>4,004</td>
<td>1</td>
</tr>
<tr>
<td>With Coron et al.’s conversion</td>
<td>313</td>
<td>63,358</td>
<td>15.82</td>
</tr>
<tr>
<td>With Biryukov et al.’s conversion</td>
<td>72</td>
<td>55,173</td>
<td>13.78</td>
</tr>
<tr>
<td>With our conversion</td>
<td>72</td>
<td>52,493</td>
<td>13.11</td>
</tr>
</tbody>
</table>

Table 3. Numbers of randomnsses and operations required for HMAC-SHA-1 with Python

From this table, the implementations of HMAC-SHA-1 with a first-order masking require more than tenfold operations compared to the one without any countermeasure. Among those with a first-order masking, our conversion leads the fast implementation. The gains against those with the Coron et al.’s and Biryukov et al.’s algorithms are 10,865 (≈ 17.1%) and 2,680 (≈ 4.9%), respectively.

As for the randomness, implementations with Biryukov et al.’s conversion and ours require 72 masks: 5 masks for the initial five words in each two hashes (subtotal 10), 16 ones for the sixteen words to each three blocks (subtotal 48), 11 ones for the remaining block (the first five words out of sixteen ones, which is an output of the inner hash, are already masked), one for the Boolean-to-arithmetic conversion, and two for the arithmetic-to-Boolean conversion.

C implementation for IPA Test Vehicle Smartcard: We then implemented HMAC-SHA-1, in C with assembly modification, with/without a countermeasure. In the implementations with countermeasures, the masks are generated by xorshift [14] with a random seed. Table 4 summarizes the numbers of cycles required for HMAC-SHA-1 in the test vehicle smartcard.

In this table, “opt0” means implementations where they are masked throughout the 80 rounds, but two results of SHA-1 compressions for the fixed keys are pre-computed. “opt1” and “opt2” mean optimized implementations where internal values in 40 and 60 middle rounds of SHA-1 are unmasked, respectively. The
<table>
<thead>
<tr>
<th>Implementation</th>
<th>#cycles</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without countermeasure</td>
<td>12,391</td>
<td>1</td>
</tr>
<tr>
<td>With Coron et al.'s conversion (opt0)</td>
<td>68,711</td>
<td>5.55</td>
</tr>
<tr>
<td>With Biryukov et al.'s conversion (opt0)</td>
<td>66,344</td>
<td>5.35</td>
</tr>
<tr>
<td>With our conversion (opt0)</td>
<td>63,546</td>
<td>5.13</td>
</tr>
<tr>
<td>With Coron et al.'s conversion (opt1)</td>
<td>41,914</td>
<td>3.38</td>
</tr>
<tr>
<td>With Biryukov et al.'s conversion (opt1)</td>
<td>40,913</td>
<td>3.30</td>
</tr>
<tr>
<td>With our conversion (opt1)</td>
<td>39,471</td>
<td>3.19</td>
</tr>
<tr>
<td>With Coron et al.'s conversion (opt2)</td>
<td>29,150</td>
<td>2.35</td>
</tr>
<tr>
<td>With Biryukov et al.'s conversion (opt2)</td>
<td>28,629</td>
<td>2.31</td>
</tr>
<tr>
<td>With our conversion (opt2)</td>
<td>27,862</td>
<td>2.25</td>
</tr>
</tbody>
</table>

Table 4. Number of cycles required for HMAC-SHA-1 in IPA test vehicle smartcard implementations with the “opt0” countermeasure require more than five times cycles compared to one without a countermeasure. If two hashes for the fixed keys are not precomputed, they should be tenfold as in Table 3. Compared to those with Coron et al.’s conversion, our gains are 5,165 (≈ 7.5%, “opt0”), 2,443 (≈ 5.8%, “opt1”), and 1,288 (≈ 4.4%, “opt2”), respectively. Compared to those with Biryukov et al.’s conversion, our gains are 2,798 (≈ 4.2%, “opt0”), 1,442 (≈ 3.5%, “opt1”), and 767 (≈ 2.7%, “opt2”), respectively.

Table 5 summarized the code sizes of C implementations. From the table, our algorithm leads the smallest code.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>size (byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Coron et al.'s conversion (opt2)</td>
<td>2,780</td>
</tr>
<tr>
<td>With Biryukov et al.'s conversion (opt2)</td>
<td>2,700</td>
</tr>
<tr>
<td>With our conversion (opt2)</td>
<td>2,640</td>
</tr>
</tbody>
</table>

Table 5. Code sizes for HMAC-SHA-1 in IPA test vehicle smartcard

4.3 Test for the First-Order Side-Channel Leakage

We implemented a software of HMAC-SHA-1 with our conversion (opt2) on the test vehicle smartcard and acquired 100,000 power consumption traces, as specified for the security level 4 in ISO/IEC 17825 [8], with a fixed key and random inputs.

HMAC-SHA-1, with a key key and a message msg, computes $\text{sha1}(\text{key} \oplus \text{opad}) || \text{sha1}((\text{key} \oplus \text{ipad}) | | \text{msg})$, where ipad and opad are the constants 0x3636 ... 36 and 0x5c5c ... 5c, respectively. As explained, in each sha1 computations, the compression of first block with the fixed key is precomputed. Our software computes
the compression of the second block with $msg$ in the inner hash, using the compression of the first block as an initial vector. We regard the output of first round in the second compression with $msg$ as an attack target.

![Graph](image1.png)

**Fig. 1.** Average traces for $\text{sha1}(msg)$ with IV = $\text{sha1}(key \oplus ipad)$

Figure 1 depicts the average traces of 1,000 first power consumption traces, where its horizontal and vertical axes represent time and power consumption (in voltage), respectively. The iterations of rounds are observed as specified in this figure. From each trace, we extract a subtrace for the first round by removing a random delay in the processing time for synchronization.

We divided the 100,000 traces into two groups by the first byte of the target, whether its Hamming weight is more than 16 or less than 16. We then applied the test from ISO/IEC 17825 [8]. The graph in Figure 2 shows the result, where its horizontal and vertical axes represent time and T-value, respectively. In this figure, there is no point which exceeds the threshold 4.5 (less than ±0.5) and we concludes that our conversion protect HMAC-SHA-1 from the first-order attack.

![Graph](image2.png)

**Fig. 2.** ISO/IEC 17825 tests for our target
5 Discussions

As we explained in Section 3.3, our algorithm requires fewer operations than the previous algorithms to convert the arithmetic mask to the Boolean one. Moreover, as shown in Section 4.3, our algorithm gives a secure implementation against the first-order side-channel attack.

5.1 Branches

Unlike the previous algorithms, ours has branches of operations based on $i$ and $n$ as in Algorithm 5. Note that, since $i$ and $n$ do not depend on the sensitive data, these branches leak no information about secret inputs. However, it increases the execution time as it is. As for the branches conditioned by $n = \max(\lceil \log_2(k - 1) \rceil, 1)$, if an architecture, especially, the size of addition $k$, is determined in advance, it is sufficient to implement a corresponding one of the operation sequences without a branch on $n$. As for the branch conditioned by $i$, on the other hand, we can remove it if the implementation of for loop is unrolled. This leads a trade-off between the time and the code size. In our previous experiment, we implemented the unrolled\(^2\) architecture in Algorithm 5.

5.2 Conversion of Higher-Order Masking

In this paper, we gave an arithmetic-to-Boolean conversion algorithm for a first-order masking. Following the discussion of [5, Section 6], our algorithm is extensible to a conversion of higher-order masking, as well as the Coron et al.’s algorithm.

6 Conclusion

In this paper, we proposed another improved conversion algorithm from the Coron et al.’s one, by reducing operations over subroutines. Our experiments, with the IPA test vehicle smartcard, showed that our conversion correctly worked as a countermeasure against the first-order attack. Discussions on sophisticated attacks such as (non-)profiled attacks and their countermeasures are our future works.

Acknowledgement

In this paper, we use the test vehicle smartcard from IPA. We would like to thank anonymous reviewers for their fruitful comments on the previous version of this manuscript.

\(^2\) Note that the implementation of SHA-1 is in the rolled architecture.
References


A Arithmetic-to-Boolean Conversion based on Biryukov et al.’s Addition

Based on the Biryukov et al.’s masked addition of Algorithm 3, an arithmetic-to-Boolean conversion is naturally derived. Algorithm 7 shows the conversion. It requires two random masks as ours does.
Algorithm 7 Arithmetic-to-Boolean Conversion based on Biryukov et al.’s Addition

Input: $A, r \in \{0, 1\}^k$ and $n = \max(\lceil \log_2(k - 1) \rceil, 1)$ such that $A = x - r \in \mathbb{F}_{2^k}$

Output: $x'$ such that $x' \oplus r = A + r \mod 2^k$

1: $x_1, y_2 \leftarrow \{0, 1\}^k$
2: $x_1 \leftarrow A \oplus x_2$
3: $y_1 \leftarrow r \oplus y_2$
4: $p_1, p_2 \leftarrow \text{SecXor2}(x_1, x_2, y_1, y_2)$
5: $g_1, g_2 \leftarrow \text{SecAnd2}(x_1, x_2, y_1, y_2)$
6: $g_1, g_2 \leftarrow ((g_1 \oplus x_2) \oplus g_2, x_2)$
7: for $i := 1$ to $n - 1$ do
8:   $h_1, h_2 \leftarrow \text{SecShift2}(g_1, g_2, 2^{i-1})$
9:   $u_1, u_2 \leftarrow \text{SecAnd2}(p_1, p_2, h_1, h_2)$
10:  $g_1, g_2 \leftarrow \text{SecXor2}(g_1, g_2, u_1, u_2)$
11:  $h_1, h_2 \leftarrow \text{SecShift2}(p_1, p_2, 2^{i-1})$
12:  $h_1, h_2 \leftarrow ((h_1 \oplus x_2) \oplus h_2, x_2)$
13:  $p_1, p_2 \leftarrow \text{SecAnd2}(p_1, p_2, h_1, h_2)$
14:  $p_1, p_2 \leftarrow ((p_1 \oplus y_2) \oplus p_2, y_2)$
15: end for
16: $h_1, h_2 \leftarrow \text{SecShift2}(g_1, g_2, 2^{n-1})$
17: $u_1, u_2 \leftarrow \text{SecAnd2}(p_1, p_2, h_1, h_2)$
18: $g_1, g_2 \leftarrow \text{SecXor2}(g_1, g_2, u_1, u_2)$
19: $x' \leftarrow A \oplus 2g_1 \oplus 2g_2$
20: return $x'$