Fast Software Encryption Attacks on AES

David Gstir and Martin Schläffer

IAIK, Graz University of Technology, Austria

martin.schlaeffer@iaik.tugraz.at

Abstract. In this work, we compare different faster than brute-force single-key attacks on the full AES in software. Contrary to dedicated hardware implementations, software implementations are more transparent and do not over-optimize a specific type of attack. We have analyzed and implemented a black-box brute-force attack, an optimized brute-force attack and a biclique attack on AES-128. Note that all attacks perform an exhaustive key search but the latter two do not need to recompute the whole cipher for all keys. To provide a fair comparison, we use CPUs with Intel AES-NI since these instructions tend to favor the generic black-box brute-force attack. Nevertheless, we are able to show that on Sandy Bridge the biclique attack on AES-128 is 17% faster, and the optimized brute-force attack is 3% faster than the black-box brute-force attack.

Keywords: fast software encryption, AES, brute-force attack, biclique attack, Intel AES-NI.

1 Introduction

In recent years, new attacks on the full Advanced Encryption Standard (AES) have been published [1,3]. Especially the single-key attacks are debatable due to their marginal complexity improvement compared to a generic exhaustive key search (brute-force). Therefore, Bogdanov et al. have implemented a variant of the biclique attack in hardware to show that their attack is indeed faster than brute-force [2].

However, in a dedicated hardware implementation it is less transparent how much effort has been put on optimizing each attack type. If the difference in complexity is very small, it may be possible to turn the result around by investing more optimization effort in the slower attack. Contrary to hardware implementations, the speed of well optimized software implementations tends to be more stable. This can also be observed when looking at different comparisons of the NIST SHA-3 candidates in hardware and in software [5,11]. Too many parameters can be optimized in hardware which can easily change a comparison in favor of one or the other primitive or attack.

In this work, we have implemented different single-key attacks on AES-128 using Intel AES-NI [7], which is the basis for the fastest software implementations of AES. We compare the generic black-box brute-force attack with an optimized

A. Youssef, A. Nitaj, A.E. Hassanien (Eds.): AFRICACRYPT 2013, LNCS 7918, pp. 359–374, 2013.
© Springer-Verlag Berlin Heidelberg 2013

brute-force attack and with the (simplified) biclique attack used in the hardware implementation of Bogdanov et al. [2]. In the optimized brute-force attack we choose keys such that we do not need to recompute the full cipher for every new key guess.

Our results indicate that the biclique attack is indeed marginally faster than a black-box brute-force attack. However, also the optimized brute-force attack on AES is slightly faster than the black-box brute-force attack. We have concentrated our effort on AES-128 but the results are likely to be similar for the other variants as well. Nevertheless, these attacks do not threaten the security of AES since for any real world cipher, optimized brute-force attacks are most likely faster than black-box brute-force attacks.

Outline of the paper: In Section 2, we give a brief description of AES-128 and the implementation characteristics of Intel AES-NI. In Section 3, we describe and evaluate the theoretical complexity of the black-box brute-force attack and an optimized brute-force attack. Section 4 describes and analyzes the simplified biclique attack on AES-128 of Bogdanov et al. [2]. Then, our software implementations and results of these three attacks are given in Section 5. Finally, we conclude our work in Section 6.

2 Implementing AES in Software Using AES-NI

In this section, we briefly describe the Advanced Encryption Standard (AES) as well as the instructions and implementation characteristics of Intel AES-NI.

2.1 Description of AES-128

The block cipher Rijndael was designed by Daemen and Rijmen and standardized by NIST in 2000 as the Advanced Encryption Standard (AES) [9]. The AES consists of a key schedule and state update transformation. In the following, we give a brief description of the AES and for a more detailed description we refer to [9].

State Update. The block size of AES is 128 bits which are organized in a 4×4 state of 16 bytes. This AES state is updated using the following 4 round transformations with 10 rounds for AES-128:

- the non-linear layer `SubBytes` (`SB`) independently applies the 8-bit AES S-box to each byte of the state
- the cyclical permutation `ShiftRows` (`SR`) rotates the bytes of row r to the left by r positions with $r = \{0, ..., 3\}$
- the linear diffusion layer `MixColumns` (`MC`) multiplies each column of the state by a constant MDS matrix
- in round i, `AddRoundKey` (`AK`) adds the 128-bit round key rk_i to the AES state

A round key rk_0 is added prior to the first round and the `MixColumns` transformation is omitted in the last round of AES. All 4 round transformations plus the enumerations of individual state bytes are shown in Fig. 1

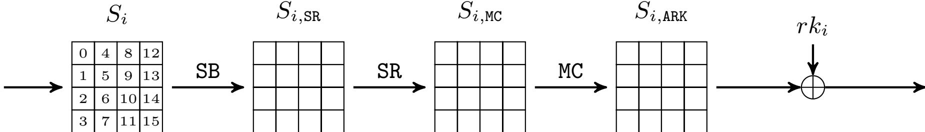

Fig. 1. Notation for state bytes and 4 round transformations of AES

Key Schedule. The key schedule of AES recursively generates a new 128-bit round key rk_i from the previous round key rk_{i-1}. In the case of AES-128, the first round key rk_0 is the 128-bit master key of AES-128. Each round of the key schedule consists of the following 4 transformations:

- an upward rotation of 4 column bytes by one position
- the nonlinear `SubWord` applies the AES S-box to 4 bytes of one column
- a linear part using XOR additions of columns
- a constant addition of the round constant `RCON[i]`

More specifically, each column $rk_{i,c}$ with $i = 1, \ldots, 10$ and $c = 0, \ldots, 3$ of round key rk_i is computed as follows:

$$rk_{i,0} = \mathtt{SubWord}(rk_{i-1,3} \ggg 1) \oplus rk_{i-1,0} \oplus \mathtt{RCON}[i] \qquad \text{for } c = 0$$
$$rk_{i,c} = rk_{i,c-1} \oplus rk_{i-1,c} \qquad \text{for } c = 1, \ldots, 3$$

2.2 Efficient Implementations of AES-128 Using AES-NI

For our software implementation we chose to use the Intel AES instruction set *AES-NI* (*AES New Instructions*) [7] because it provides the fastest way to implement AES on a standard CPU and makes the implementations easier to compare. Moreover, AES-NI gave us a fair basis for the brute-force and biclique implementations since all AES operations take exactly the same time throughout all implementations.

For the AES-NI instruction set, Intel integrated certain operations for AES directly into the hardware, thus, making them faster than any pure software implementation and providing more security against timing attacks due to constant time operations [7]. Overall, AES-NI adds the following operations for key schedule, encryption and decryption and a full description of all instructions can be found in [8]:

- `aesenc`, `aesdec` performs one full encryption or decryption round, respectively.
- `aesenclast`, `aesdeclast` performs the last encryption or decryption round.
- `aeskeygenassist` computes the `SubWord`, rotation and `XOR` with `RCON` operations required for the key schedule.
- `aesimc` performs `InvMixColumns` on the given 128-bit register and stores the result to another 128-bit register.

Modern CPUs use multiple techniques to boost performance of applications. Hence, creating software implementations with optimal performance requires some background knowledge on how CPUs operate. For our software implementations we took the following approaches into account to increase the performance:

High Pipeline Utilization: CPUs split a single instruction into multiple μops. This enables the CPU to start processing the next instruction before the previous has finished. Obviously, this only works if both instructions are independent of each other.

Minimal Memory Access: Fetching data from memory (registers) outside the CPU is slow and decreases performance. Since the attacks shown here, have minimal memory complexities, they can be implemented without almost any memory access.

Parallelized Encryption: For optimal pipeline utilization it is important to carefully utilize AES-NI and SSE instructions since they normally take more than one CPU cycle. Therefore, we compute each encryption round for multiple keys at the same time. This results in multiple independent instructions which are processed by the CPU. This in turn leads to higher pipeline utilization. For instance, on Intel Westmere the execution of `aesenc` takes 6 CPU cycles and the CPU can execute an instruction every second cycle. If we perform 4 independent instructions in parallel, it would require 6 cycles until the first operation is finished. After that, every second cycle another operation finishes. So, in total it requires only 12 cycles for those four operations to finish, instead of 24 cycles if we do not parallelize them. On Intel Sandy Bridge, one execution of `aesenc` takes 8 CPU cycles and the CPU can execute an instruction every cycle.

Reducing `aeskeygenassist` Instructions: The `aeskeygenassist` instruction performs suboptimal on current Sandy Bridge CPUs. Our tests have shown that this instruction has a latency of 8 cycles and a reciprocal throughput of 8 (independently shown in [4]). This is slower compared to the other AES-NI instructions. Since we have to compute the key schedule very often in all attacks, we need to compute round keys as fast as possible. Fortunately, `aeskeygenassist` is able compute to `SubWord` and the rotation for two words in parallel. Thus, we use a single `aeskeygenassist` instruction for two independent round keys. Another solution would be to avoid `aeskeygenassist` and compute `SubWord` using `aesenclast` and the rotation using an SSE byte shuffle instruction.

3 Brute-Force Key Recovery Attacks on AES-128

In this section we describe two brute-force attacks on AES-128 and evaluate their complexities. We show that an optimized brute-force attack which does not recompute all state bytes is in theory (marginally) faster than a black-box brute-force attack. A practical evaluation of these two attacks in software using AES-NI is given in Section 5.

3.1 Black-Box Brute-Force Attack

We call a generic key recovery attack which does not exploit any structural properties of a cipher a black-box brute-force attack. This is the only possible key recovery attack on an ideal cipher. For a cipher with key size n, 2^n keys have to be tested to find the unknown encryption key with probability 1. Thus, the generic complexity is determined by 2^n evaluations of the cipher. In such an attack, the data complexity is 1 and the key complexity is 2^n. Note that time-memory trade-offs apply if more keys are attacked at the same time [6], while the black-box brute-force attack and biclique attack need to be repeated for each key to attack. To compare a black-box brute-force attack on AES-128 with an optimized brute-force attack or any other brute-force attack, we need to determine its complexity in terms of AES-128 evaluations. Since most optimized attacks compute only parts of the cipher, we evaluate the complexity in terms of S-box computations, the most expensive part of most AES implementations. In total, one full AES-128 encryption including key schedule computation requires to compute 200 S-boxes.

3.2 Optimized Brute-Force Attack

Every practical cipher consists of non-ideal sub-functions or rounds. If the diffusion is not ideal, a flip in a single key bit does not immediately change all bits of the state. This effect can be exploited by an optimized brute-force attack. Instead of randomly testing keys in a key recovery attack, we can iterate the keys in a given order, such that only parts of the cipher need to be recomputed for each additional key. Hence, we save computations which may reduce the overall cost of performing a brute-force key recovery attack. Note that every optimized brute-force attack still needs to test all 2^n keys, which is not the case in a short-cut key recovery attack.

Such an optimized brute-force attack is possible for every real world cipher. However, the complexity reduction will only be marginal. In practical implementations of an attack it can even be worse, since computing a full round is usually more efficient than computing, extracting, restructuring and combining parts of a computation (see Section 5). After all, the final result depends heavily on the implementations and how well they are optimized themselves. Nevertheless, in the case of AES we can still count and compare S-box computations as an estimate for the optimized brute-force complexity.

In the following, we give a basic example of an optimized brute-force attack on AES-128. Instead of trying keys randomly, we first iterate over all values of a single key byte and fix the remaining 15 key bytes. Hence, we only compute the whole cipher once for a base key, and recompute only those parts of the state which change when iterating over the 2^8 values for a single byte. Fig. 2 shows those bytes in white, which do not need to be recomputed for every key candidate. The number of white bytes also roughly determines the complexity reduction compared to a black-box brute-force attack. To save additional recomputations on the last two rounds, we match the ciphertext only on four instead of all 16 bytes.

For each set of 2^8 keys, we save the computation of 15 S-boxes of state S_1, 9 S-boxes of state S_2, 12 S-boxes of state S_9 and 12 S-boxes of state S_{10}, in total 48 S-boxes. In the key schedule, we save the computation of 4 S-boxes in rk_0, 3 S-boxes in rk_1, rk_2 and rk_{10}, 2 S-boxes in rk_3 and 1 S-box in rk_4 in total 16 S-boxes. Hence, instead of 200 S-boxes we need to compute only 136 S-boxes or 0.68 full AES-128 evaluations. Therefore, the total complexity of this optimized brute-force attack is about

$$2^{120} \cdot (1 + 255 \cdot 0.68) = 2^{127.45}$$

full AES-128 evaluations.

4 Simplified Biclique Attack for Hardware Implementation

To evaluate the biclique attack [3] in hardware, Bogdanov et al. have proposed a simplified variant of the biclique attack [2] which is more suitable for practical implementations. To verify that the attack is practically faster than a black-box brute-force attack, they modified the biclique attack to reduce its data complexity and simplified the matching phase, where the full key is recovered. The main difference to the original biclique attack is that only a 2-dimensional biclique is applied on the first two rounds instead of the last three rounds. Furthermore, the key recovery phase matches on four bytes of the ciphertext instead of some intermediate state. This simplifies the matching phase since it is basically just a forward computation from state S_3 to these four bytes of the ciphertext. In this section, we briefly cover the theory of the modified attack from [2] and give a comparison of each step to a black-box brute-force attack on AES.

4.1 Biclique Construction

The modified biclique attack targets AES-128. The biclique originates from the idea of initial structures [10] and is placed on the initial two rounds and the meet-in-the-middle matching to recover the encryption key is done on the remaining eight rounds. The key space is divided into 2^{124} groups of 2^4 keys each. These key groups are constructed from the initial cipher key rk_0 (whitening key) and do not overlap. The partitioning of the key space defines the dimension d of the biclique which is $d = 2$.

Each key group is constructed from a base key. We retrieve the base keys by setting the two least significant bits of $rk_0[0]$ and $rk_0[6]$ to zero and iterating the remaining bits of rk_0 over all possible values. Hence, bytes 0 and 6 of the base key have the binary value $b = b_0b_1b_2b_3b_4b_500$. To get the 16 keys within a key group, we enumerate differences $i, j \in \{0, \ldots, 3\}$ and add them to bytes 0, 4, 6, and 10 of the base key (see rk_0 in Fig. 3). Note, that we add the same difference i to bytes 0 and 4 as well as difference j to bytes 6 and 10. This is done to cancel

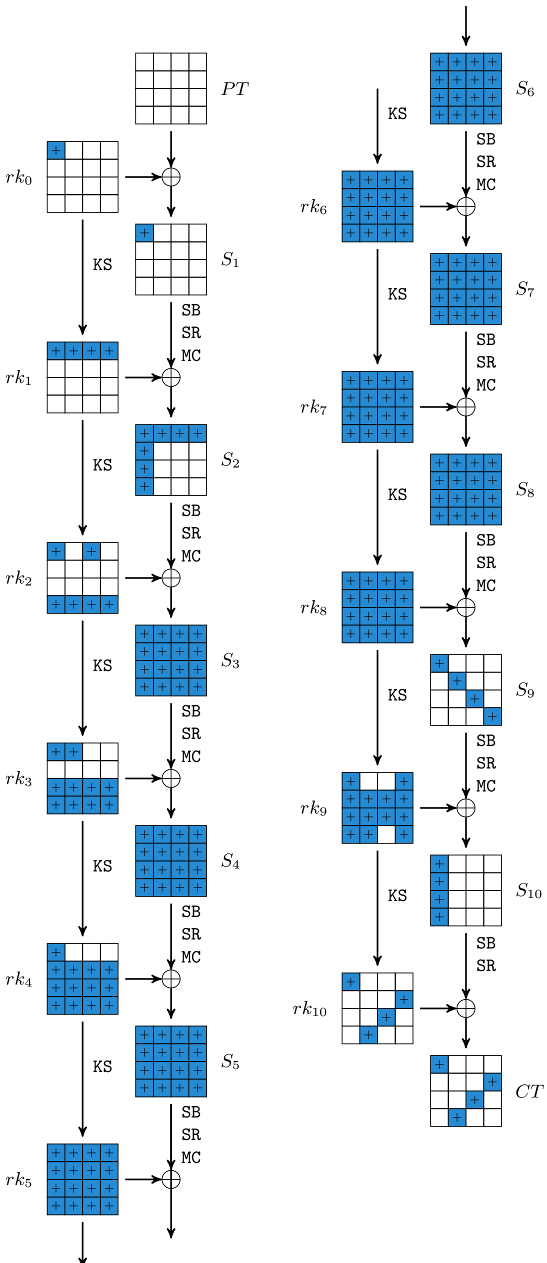

Fig. 2. Partial states which have to be recomputed for each set of 2^8 keys in the optimized brute-force attack. Only bytes ⊞ (input to S-boxes) need to be recomputed for every key guess. All empty bytes are recomputed only once for each set of 2^8 keys. In total, we save the computation of 48 S-boxes in the state update and 16 S-boxes in the key schedule.

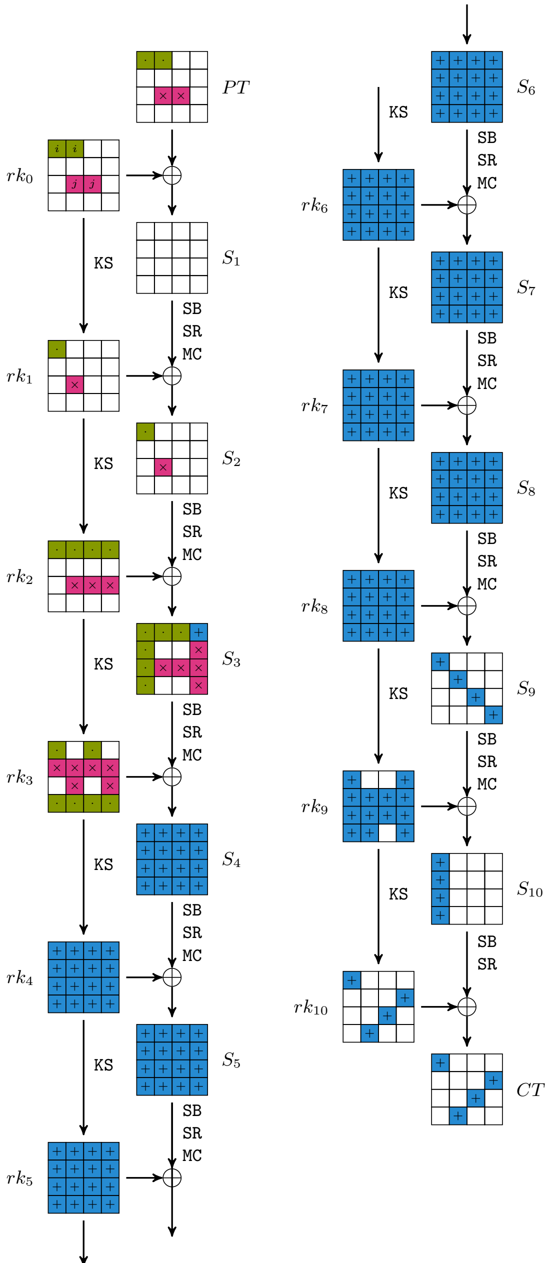

Fig. 3. Partial states which have to be recomputed for each set of 2^{16} keys in the hardware biclique attack. Blue bytes (+) need to be recomputed for every key guess. White bytes are recomputed only for each set of 2^{16} keys. Bytes indicated by · and × are used in the biclique structure and need to be recomputed 7 times for each set of 2^{16} keys.

some differences in the following round key such that only bytes 0 and 6 of rk_1 have non-zero differences.

Similar to the original biclique attack, we use these key differences to construct two differential trails Δ_i and ∇_j. The Δ_i-trail is constructed from the key difference Δ_i^K which covers all keys with $i \in \{1, \ldots, 3\}$ and $j = 0$. For the key difference ∇_j^K (∇_j-trail), we fix $i = 0$ and enumerate $j \in \{1, \ldots, 3\}$. Additionally, the key differences are also used as plaintext differences for the respective trail.

Finally, to construct the biclique, we combine both trails as shown in Fig. 3. This yields a mapping of 16 plaintexts to 16 values for the intermediate state S_3 under the 16 keys of a group[1]. Since both differential trails are not fully independent (they both have byte 12 active in S_3), we have to consider and recompute this byte separately as described below.

Until here, we have constructed the biclique from two (almost) independent differential trails. This yields 16 values for S_3 for the 16 plaintexts encrypted under the corresponding key from a key group. Thus, for each key group, we can retrieve a different set of 16 values for S_3. This enables an effective computation of 16 values for S_3 by performing the following steps for each key group:

1. Perform the base computation by taking the all-zero plaintext and encrypting it with the base key of the group. Store the resulting value for S_3 (S_3^0) and the value for rk_2 (rk_2^0).
2. Enumerate the difference $i \in \{1, 2, 3\}$, set $j = 0$ and recompute the active byte of the combined differential trail (indicated by ▪ in Fig. 3). This yields three values for S_3, denoted by S_3^i.
3. Perform similar computations for $j \in \{1, 2, 3\}$ and $i = 0$ to get three more values for S_3, denoted by S_3^j (indicated by ☒).
4. Combine the values for S_3^0 from the base computation, S_3^i and S_3^j to get the 16 values for S_3[2].
5. Since $S_3[12]$ is active in both differential trails, we consider this byte separately and retrieve its value by calculating $S_3^j[12] \oplus i$ or alternatively $S_3^i \oplus j$.

The advantage of the biclique construction over a black-box brute-force attack is that we save S-box computations by simply combining precomputed values for S_3. For a black-box brute-force attack, we perform 16 3-round AES computations to get 16 values for S_3. Here, we compute the full three rounds only for the base computation and then recompute only the required bytes for each differential trail for $i, j \in \{1, 2, 3\}$.

There is one possible improvement that enables us to save some S-box evaluations in the matching phase: Instead of computing values for S_3, we can include the following `SubBytes` and `ShiftRows` operations and compute $S_{3,\texttt{MC}}$ (the state after `MixColumns` in round 3) instead. In the base computation, we compute the

[1] Note that this does not exactly match the definition of a biclique as it maps 2^{2d} plaintexts to 2^{2d} states under 2^{2d} keys.

[2] To get the values for rk_2, we just add the i, j differences to the corresponding bytes of rk_2^0. This is possible because no S-box has to be computed for the active key bytes in the key schedule.

S-box for bytes 5,7,9,11, and for each of the differential trails we compute the S-box for their corresponding active bytes. This leaves only byte 12, which we have to compute for all 16 possible values of $S_{3,\texttt{MC}}$ (see also [2]).

4.2 Key Recovery

As already described, the matching phase is simplified to match on four bytes of the ciphertexts. Thus, we first take the 16 plaintexts and retrieve the corresponding ciphertexts from the encryption oracle (This has to be performed only once for the full attack). From the output of the biclique (16 values for $S_{3,\texttt{MC}}$), we then compute only the required bytes to get the four ciphertext bytes. As shown in Fig. 3, we compute the full states from S_4 up to $S_{7,\texttt{MC}}$. From this state on until the ciphertext, we only compute reduced rounds. For the resulting four ciphertext bytes, we check if they match the corresponding ciphertext bytes retrieved from the encryption oracle. If they do, we have a possible key candidate which we have to verify with one additional plaintext-ciphertext pair. However, we should get only about one false key candidate per 2^{32} keys.

The advantage of this phase over a black-box brute-force attack is that we avoid some S-box computations by matching on only four bytes of the ciphertext instead of all 16 bytes. Note that this part is identical to the optimized brute-force attack described in Section 3.2.

4.3 Complexity of the Attack

The data complexity is defined by the biclique and is thus 2^4. Concerning the time complexity, Bogdanov et al. estimated the computation of the 16 values for S_3 using the biclique (precomputation phase) to be at most 0.3 full AES-128 encryptions. For the matching phase they estimated an effort similar to 7.12 AES-128 encryptions. Thus, the time complexity is

$$2^{124} \cdot (0.3 + 7.12) = 2^{126.89}$$

AES-128 encryptions.

However, our calculation yields a slightly higher complexity: The biclique computation requires 8 S-boxes for key schedule up to rk_2, 32 S-boxes for the base computation and 6 S-boxes for the recomputations of both trails. Moreover, for computing $S_{3,\texttt{MC}}$, we have to compute 4 S-boxes for the inactive bytes, $4 \times 11 = 44$ for the active bytes of both differential trails in round 3, and 16 S-box evaluations for computing byte 12. This results in a total of 110 S-boxes. Computing three AES rounds (incl. the key schedule) for 16 keys, as it would be done in a brute force attack, takes $16 \times 3 \times 20 = 960$ S-Box computations. Thus, the precomputation phase is about the same as 0.11 3-round AES-128 encryptions or equivalently 0.55 full AES-128 executions.

The matching phase has to be performed for every one of the 16 keys in a key group. This phase requires 80 S-boxes for computing the full rounds 4-8, 8 S-boxes for the last two rounds and 29 S-boxes for the key schedule. This

results in a total of $16 \times 117 = 1872$ S-box evaluations per key group. Since a brute-force attack for 16 keys on seven rounds requires $16 \times 7 \times 20 = 2240$ S-box computations, the matching phase is about the same as 0.84 7-round AES-128 encryptions or equivalently 9.36 full AES-128 executions. The resulting time complexity of the full modified biclique attack is thus

$$2^{124} \cdot (0.55 + 9.36) = 2^{127.31}.$$

5 Software Implementations and Benchmark Results

To compare the two brute-force attacks with the biclique attack, we have implemented all three variants in software using AES-NI. Of course, GPU or dedicated hardware implementations of these attacks will always perform better and are less costly than software implementations on standard CPUs. However, we use these software implementations to provide a more transparent and fair comparison of the attacks. The efficiency of hardware implementations depends a lot on the used device or underlying technology, which may be in favor of one or the other attack. Moreover, which attack performs better also depends a lot on the effort which has been spent in optimizing the attack.

In the case of software implementations using AES-NI all attacks have the same precondition. If AES-NI benefits one of the attacks, it is the black-box brute-force attack which needs to compute only complete AES rounds. In the following, we will show that nevertheless, both the optimized brute-force attack as well as the biclique attack are slightly faster than the generic attack.

For each attack, we have created and benchmarked a set of implementations to rule out less optimized versions:

- assembly implementations and C implementation using intrinsics
- parallel versions using 4x and 8x independent AES-128 executions
- benchmarked on Intel Westmere and Intel Sandy Bridge CPUs

Since Intel AES-NI has different implementation characteristics on Westmere and Sandy Bridge, we get slightly different results but the overall ranking of attacks does not change. We have also tried to used AVX instructions to save some `mov` operations. However, this has only a minor impact on the results. Note that in the C intrinsics implementations the compiler automatically uses AVX if it is available on the target architecture.

5.1 Black-Box Brute-Force Implementation

The implementation of the black-box brute-force attack is quite straightforward. Nevertheless, to find the fastest implementation we have evaluated several approaches. We have implemented two main variants, which test eight or four keys in parallel. In the 8x parallel variant, the 6-cycle (Westmere) or 8-cycles (Sandy Bridge) latency can be hidden. However, we need more memory accesses compared to the 4x variant, since we cannot store all keys, states and temporary

values in the 16 128-bit registers. Therefore, the 4x variant may be faster in some cases.

The main bottleneck of AES implementations using AES-NI is the rather slow `aeskeygenassist` instruction. Therefore, the implementations do not reach the speed given by common AES-NI benchmarks without key schedule recomputations. Since the throughput of the instruction `aeskeygenassist` is much lower than the `aesenc` instruction, we compute the key schedule on-the-fly. This way, we also avoid additional memory operations.

The full performance measurements for all implementations are shown in Table 1. The 8x variants test eight keys in parallel but require memory access, the 4x variants test four keys in parallel without any memory access. The table shows nicely that the memory-less implementation can in fact be faster under certain circumstances. Overall, the performance of an implementation depends highly on the latency of the AES instructions. E.g. on the Sandy Bridge architecture, the instructions take longer and testing only four keys in parallel does not utilize the CPU pipeline optimally.

Table 1. Performance measurements for the various software implementations of the *black-box brute-force attack*, *optimized brute-force attack* and *biclique attack*. All values are given in cycles/byte. Best results per architecture and implementation are written in bold.

Approach	Black-Box Brute-F.		Optimized Brute-F.		Biclique (Modified)	
	Westmere	Sandy B.	Westmere	Sandy B.	Westmere	Sandy B.
C, 4x	3.09	**3.80**	3.20	3.70	2.71	**3.18**
C, 4x, rnd 1 full			**3.10**	3.75		
ASM, 4x	**3.00**	**3.80**			**2.61**	3.24
ASM-AVX, 4x		**3.80**				3.21
C, 8x	3.45	3.86	3.52	3.89	3.41	3.39
C, 8x, rnd 1 full			3.24	**3.67**		
ASM, 8x	3.40	3.95				
ASM-AVX, 8x		3.93				

5.2 Optimized Brute-Force Attack Implementation

The idea for the optimized brute-force attack is to avoid some computations by iterating all possible keys in a more structured way. As we have seen in Section 3.2, this slightly reduces the time complexity of the attack. To verify this, we implemented multiple variants of this attack (4 and 8 keys in parallel) using AES-NI.

However, since Intel AES-NI is optimized for computing full AES rounds, it is not possible to efficiently recompute only the required bytes and S-boxes.

Hence, we often compute the full state although we only need some parts of it. This is for instance the case for round 2 (see Fig. 2). The additional instructions required to perform only a partial encryption in this round take longer than just using `aesenc` to compute the full encryption round.

Table 1 lists the full measurements for this attack. We also included a comparison between computing the full round 1 and computing only the reduced round as it is the idea for this attack. The results clearly show that it is also faster (in most cases) to just compute the full round instead of a reduced round.

Nevertheless, the last 2 rounds can be computed as reduced rounds since one round does not contain `MixColumns` which makes the rearrangement of states less complex. For these last 2 reduced rounds we collect the required bytes of four states into one 128-bit register and use one `aesenc` (or `aesenclast`) instruction to compute the remaining bytes. In the matching phase, we save computations in the last two rounds. For testing 4 keys, we need to compute only 2 AES rounds and some recombinations instead of $4 \cdot 2$ AES rounds.

5.3 Biclique Attack Implementation

The modified biclique attack by Bogdanov et al. as covered in Section 4 has several advantages when implemented in software. Most notable are the low data complexity of only 16 plaintext-ciphertext pairs and the simple matching phase at the end of the cipher on four ciphertext bytes. These modifications allow us to implement the attack with almost no memory operations.

To better exploit the full-round `aesenc` instruction, we compute the biclique only until state S_3 and not $S_{3,\mathtt{MC}}$. Since the biclique attack considers groups of 16 keys, our main loop consists of three steps:

1. Precompute values for S_3^0, rk_2 and active bytes of S_3^i, S_3^j.
2. Combine the precomputed bytes to 16 values for the full state S_3.
3. Encrypt the remaining rounds and match with the ciphertexts from the encryption oracle.

For the biclique computation we basically follow the steps given in Section 4.1. However, to compute the differential trails, we directly compute the combined trail for $i = j \in \{1, 2, 3\}$. Afterwards, we extract the values for S_3^i and S_3^j to construct 16 values for S_3 using the base computation. Consequently, we perform 4 full computations of the first 2 rounds per key group. This phase is equal for the 4x and 8x variants.

The 5 full-round computations of the third step are exactly the same as in the black-box brute-force attack. The 3 round computations at the end and the matching are the same as in the optimized brute-force attack. Similar to these attacks, we have implemented two variants, which compute and match either 4 or 8 keys in parallel. Again, we compute the key schedule on-the-fly since this results in faster code.

In general, our implementations of the biclique attack are quite similar to the brute-force attacks. Similar to the optimized brute-force attack, we also compute

full round more often than necessary. For example, in round 3 and round 8 we compute full rounds although we would not need to. Especially computing only `SubBytes` by `aesenclast` followed by `pshufb` is more expensive than computing a whole round.

Overall, for the full matching phase (4x and 8x variants) we compute six full rounds with `aesenc` and then compute the remaining two rounds as reduced rounds. As can be seen in Fig. 2 and Fig. 3 the reduced rounds of the biclique attack and the optimized brute-force attack are in fact equal. Consequently, the implementation of the matching phase is equal to the last 8 rounds of the optimized brute-force attack.

5.4 Performance Results

We have tested our implementations of all three attacks on Intel Westmere (MacBook Pro with Intel Core i7 620M, running Ubuntu 11.10 and gcc 4.6.1) and Intel Sandy Bridge (Google Chromebox with Intel Core i5 2450M, running Ubuntu 12.04 and gcc 4.6.3). For the C implementations we used the following compiler flags: `-march=native -O3 -finline-functions -fomit-frame-pointer -funroll-loops`. All results are shown in Table 1.

Overall, the biclique attack is 13% faster on Westmere and 17% faster on Sandy Bridge, compared to the best black-box brute force on the same architecture. This clearly verifies that the biclique attack is faster than both brute-force attacks in all scenarios, although the advantage is smaller than in theory. Note that the Sandy Bridge implementations are slower in general but provide a larger advantage over the black-box brute-force attack.

The performance of the optimized brute-force attack varies depending on the CPU architecture and is actually slower than the black-box brute-force attack on Westmere CPUs. On Sandy Bridge, the optimized brute-force attack is 3% faster than the black-box brute-force attack. However, assembly implementations of the optimized brute-force attack may slightly improve the results.

If we compare Sandy Bridge implementations, the biclique attack results in a time complexity of about $2^{127.77}$ full AES-128 computations. For the optimized brute-force attack we get a complexity of $2^{127.95}$ in the best case. In the theoretical evaluation of the modified biclique attack, we have estimated an average complexity of 9.9 AES-128 encryptions to test 16 keys. However, using AES-NI we are able to get a complexity of only 14 AES-128 encryptions to test 16 keys.

6 Conclusions

In this work, we have analyzed three different types of single-key attacks on AES-128 in software using Intel AES-NI. The first attack is the black-box brute-force attack with a generic exhaustive key search complexity of 2^{128} AES computations. We have used this implementation as the base line for a comparisons of other attacks faster than brute-force. We get the best advantage of the faster

than brute-force attacks on Sandy Bridge CPUs. In this case, the simplified biclique attack by Bogdanov et al. is 17% faster than the black-box brute-force attack, while our simple optimized brute-force attack is only 3% faster.

Note that we did not put much effort in the optimized brute-force attack. More clever tricks, better implementations or using a different platform may still improve the result. Nevertheless, neither the optimized brute-force attack nor the biclique attack threaten the security of AES in any way, since still all 2^n keys have to be tested. In this sense, both attacks just perform an exhaustive key search in a more or less optimized and clever way.

With this paper, we hope to provide a basis for a better comparison of close to brute force attacks. The open problem is to distinguish between clever brute-force attacks and worrisome structural attacks which may extend to attacks with less than a marginal improvement over the generic complexity. Based on our implementation and analysis, the biclique attack in its current form is probably not such an attack. Therefore, we believe that trying to improve the biclique attack itself is more important than merely applying it to every published cipher.

Acknowledgements. We thank Vincent Rijmen for valuable comments. This work has been supported in part by the Secure Information Technology Center-Austria (A-SIT) and by the Austrian Science Fund (FWF), project TRP251-N23.

References

1. Biryukov, A., Khovratovich, D.: Related-Key Cryptanalysis of the Full AES-192 and AES-256. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 1–18. Springer, Heidelberg (2009)
2. Bogdanov, A., Kavun, E.B., Paar, C., Rechberger, C., Yalcin, T.: Better than Brute-Force Optimized Hardware Architecture for Efficient Biclique Attacks on AES-128. In: Workshop records of Special-Purpose Hardware for Attacking Cryptographic Systems – SHARCS 2012, pp. 17–34 (2012), `http://2012.sharcs.org/record.pdf`
3. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique Cryptanalysis of the Full AES. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 344–371. Springer, Heidelberg (2011)
4. Fog, A.: Instruction tables – Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs (2012), `http://www.agner.org/optimize/instruction_tables.pdf` (accessed September 2, 2012)
5. Gaj, K.: ATHENa: Automated Tool for Hardware EvaluatioN (2012), `http://cryptography.gmu.edu/athenadb/fpga_hash/table_view` (accessed February 1, 2013)
6. Hellman, M.E.: A cryptanalytic time-memory trade-off. IEEE Transactions on Information Theory 26(4), 401–406 (1980)
7. Intel Corporation:ntel® Advanced Encryption Standard (AES) Instruction Set, White Paper. Tech. rep., Intel Mobility Group, Israel Development Center, Israel (January 2010)

8. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (March 2012)
9. NIST: Specification for the Advanced Encryption Standard (AES). National Institute of Standards and Technology (2001)
10. Sasaki, Y., Aoki, K.: Finding Preimages in Full MD5 Faster Than Exhaustive Search. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 134–152. Springer, Heidelberg (2009)
11. SHA-3 Zoo Editors: SHA-3 Hardware Implementations (2012), `http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations` (accessed February 1, 2013)