Secure and Fast Implementations of Two Involution Ciphers

Billy Bob Brumley*

Aalto University School of Science and Technology, Finland
billy.brumley@tkk.fi

Abstract. Anubis and Khazad are closely related involution block ciphers. Building on two recent AES software results, this work presents a number of constant-time software implementations of Anubis and Khazad for processors with a byte-vector shuffle instruction, such as those that support SSSE3. For Anubis, the first is serial in the sense that it employs only one cipher instance and is compatible with all standard block cipher modes. Efficiency is largely due to the S-box construction that is simple to realize using a byte shuffler. The equivalent for Khazad runs two parallel instances in counter mode. The second for each cipher is a parallel bit-slice implementation in counter mode.

Keywords: Anubis, Khazad, involution ciphers, block ciphers, software implementation, timing attacks.

1 Introduction

Anubis and Khazad are two block ciphers by Barreto and Rijmen submitted during the NESSIE project (see [12] for a summary). Anubis [2] works on 128-bit blocks and is quite similar in many respects to AES. Khazad [3] is a "legacy-level" cipher working on 64-bit blocks and is closely related to Anubis. These are both involution ciphers: decryption differs from encryption only in the key schedule.

The motivation for this work comes largely from cache-timing attacks, where an attacker attempts to recover parts of the cryptosystem state by observing the variance in timing measurements due to processor data caching effects. These attacks can be time-driven and carried out remotely by measuring the latency of a high level operation, or trace-driven and locally by exploiting the cache structure to determine the sequence of lookups the cryptosystem performs. The vulnerability exists when part of the state is used as an index into a memory-resident table.

A high-speed table-based implementation of AES unrolls lower level operations such as SubBytes, ShiftRows, and MixColumns into four tables of size 256 containing 32-bit values. Lookups into these tables, indexed by state values,

* Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).

are combined with XOR to carry out AES rounds in a more software-friendly manner, relaxing the need to manipulate a large number of single byte values and bits within those bytes. Similar versions of both Anubis and Khazad exist, in fact provided as the C reference implementations and discussed in both specifications [2, 3, Sect. 7.1].

Cache-timing attacks are a serious threat and can easily lead to leakage of key material. Although there are numerous published attacks on such implementations, a practical noteworthy one is Bernstein's AES time-driven attack [5]. Anubis and Khazad are presumably susceptible to this and other timing attacks. In light of these attacks, a reasonable security requirement for any cipher is that it can be implemented to use a constant amount of time. In this context, Bernstein defines constant as "independent of the AES key and input" [5, Sect. 8]. The concept of security within this paper is with respect to timing attacks.

To this end, this work shows that constant-time and efficient implementations of both Anubis and Khazad are possible. Four such implementations appear herein, summarized as follows.

- The first Anubis implementation runs only one instance of the cipher, compatible with all standard block cipher modes. This is efficient due to a byte-vector shuffle instruction, allowing elegant realization of the nonlinear layer in constant-time. The Khazad implementation is otherwise analogous but with a smaller state runs two parallel cipher instances, here in counter mode under the same key.
- The second Anubis implementation bit-slices the state and runs eight parallel instances, here in counter mode. Not surprisingly, this is faster but requires a parallel block cipher mode. Analogously, the Khazad implementation runs 16 parallel instances.

This work builds upon two recent results on AES software implementations that remarkably manage to achieve constant-time and exceptional performance at one stroke.

- A common hardware technique to compute the AES S-box uses an isomorphism $\mathbb{F}_{2^8} \rightarrow \mathbb{F}_{2^4}^2$ and subsequently reduces the problem of inversion in the latter field to that of one in in the ground field; [13, 14, 7] are good examples of this. Using a similar technique in software when equipped with a byte-vector shuffle instruction and using a novel field element representation, Hamburg presents techniques for fast and constant-time software implementation of AES [10]. Running only a single instance of the cipher, the implementation is compatible with all standard block cipher modes.
- Käsper and Schwabe present AES bit-slice techniques, aligning individual bits of state bytes in distinct registers [11]. The implementation runs eight parallel streams in counter mode under the same key. Not only does this provide a constant-time implementation, but also is currently the fastest published AES counter mode implementation in software (not considering newer Intel models equipped with AES instruction set extensions). Table-based AES implementations on common platforms that can perform only

one load instruction per cycle are inherently limited to ten cycles per byte; there are ten rounds requiring sixteen lookups each. The authors show that bit-slicing circumvents this limit.

2 Cipher Descriptions

This section gives a description of the Anubis and Khazad primitives, specifically each component of the ciphers. The ciphers share some components verbatim and others only differ slightly. The notation here follows style of the specifications; see them for a more formal treatment [2, 3].

2.1 The Anubis Cipher

Although Anubis supports variable length keys, this work only explicitly considers 16-byte keys; generalizations are straightforward. Analogous to AES-128, Anubis consists of a 16-byte state. The state is either viewed as a vector in $\mathbb{F}_{2^8}^{16}$ or a 4×4 matrix with entries in $\mathbb{F}_{2^8}$ depending on the context. The specification denotes this by a map μ, but this work omits this formalization; flattening the matrix row-wise (concatenating the rows) yields the vector representation.

The nonlinear layer γ This layer is otherwise analogous to the AES SubBytes step, but with a different S-box. It applies an S-box $S : \mathbb{F}_{2^8} \to \mathbb{F}_{2^8}$ to each byte of the input. To facilitate efficient hardware implementation, the designers chose to build S using a three layer substitution-permutation network (SPN), where each layer includes two S-boxes $P, Q : \mathbb{F}_{2^4} \to \mathbb{F}_{2^4}$ termed "mini-boxes". Fig. 1 depicts this structure.

The transposition τ Viewing the input as a 4×4 matrix, this mapping outputs the transpose. To illustrate:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & A & B \\ C & D & E & F \end{bmatrix} \mapsto \begin{bmatrix} 0 & 4 & 8 & C \\ 1 & 5 & 9 & D \\ 2 & 6 & A & E \\ 3 & 7 & B & F \end{bmatrix}.$$

The linear diffusion layer θ This layer shares some similarities with the AES MixColumns step. It multiplies the input in matrix form by the symmetric matrix

$$H = \begin{bmatrix} 01 & 02 & 04 & 06 \\ 02 & 01 & 06 & 04 \\ 04 & 06 & 01 & 02 \\ 06 & 04 & 02 & 01 \end{bmatrix} = \begin{bmatrix} 1 & x & x^2 & x^2+x \\ x & 1 & x^2+x & x^2 \\ x^2 & x^2+x & 1 & x \\ x^2+x & x^2 & x & 1 \end{bmatrix}$$

and $\theta : a \mapsto a \cdot H$ with all operations done in $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8+x^4+x^3+x^2+1)$.

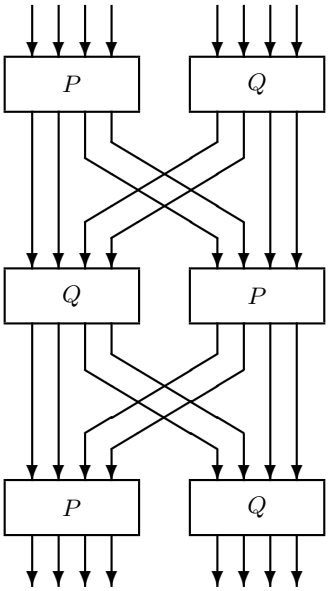

Fig. 1. S-box S as a three layer SPN with mini-boxes P and Q.

The cyclical permutation $\boldsymbol{\pi}$ This operation is otherwise analogous to the AES ShiftRows step, but cyclically shifts column i of the matrix downward i positions instead. This map only appears in the key schedule. To illustrate:

$$\begin{bmatrix} \mathtt{0} & \mathtt{1} & \mathtt{2} & \mathtt{3} \\ \mathtt{4} & \mathtt{5} & \mathtt{6} & \mathtt{7} \\ \mathtt{8} & \mathtt{9} & \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} & \mathtt{E} & \mathtt{F} \end{bmatrix} \mapsto \begin{bmatrix} \mathtt{0} & \mathtt{D} & \mathtt{A} & \mathtt{7} \\ \mathtt{4} & \mathtt{1} & \mathtt{E} & \mathtt{B} \\ \mathtt{8} & \mathtt{5} & \mathtt{2} & \mathtt{F} \\ \mathtt{C} & \mathtt{9} & \mathtt{6} & \mathtt{3} \end{bmatrix}.$$

The key extraction $\boldsymbol{\omega}$ This is a linear mapping involving the Vandermonde matrix

$$V = \begin{bmatrix} \mathtt{01} & \mathtt{01} & \mathtt{01} & \mathtt{01} \\ \mathtt{01} & \mathtt{02} & \mathtt{02}^2 & \mathtt{02}^3 \\ \mathtt{01} & \mathtt{06} & \mathtt{06}^2 & \mathtt{06}^3 \\ \mathtt{01} & \mathtt{08} & \mathtt{08}^2 & \mathtt{08}^3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & x & x^2 & x^3 \\ 1 & x^2+x & x^4+x^2 & x^6+x^5+x^4+x^3 \\ 1 & x^3 & x^6 & x^5+x^4+x^3+x \end{bmatrix} = \begin{bmatrix} \mathtt{01} & \mathtt{01} & \mathtt{01} & \mathtt{01} \\ \mathtt{01} & \mathtt{02} & \mathtt{04} & \mathtt{08} \\ \mathtt{01} & \mathtt{06} & \mathtt{14} & \mathtt{78} \\ \mathtt{01} & \mathtt{08} & \mathtt{40} & \mathtt{3A} \end{bmatrix}$$

and $\omega : a \mapsto V \cdot a$. This map also only appears in the key schedule.

The key schedule Given the cipher key K, round keys K^i for $0 \leq i \leq 12$ satisfy $K^r = (\tau \circ \omega \circ \gamma)(\kappa^r)$ where $\kappa^0 = K$ and $\kappa^r = (\sigma[c^r] \circ \theta \circ \pi \circ \gamma)(\kappa^{r-1})$ for $r > 0$, σ is addition in $\mathbb{F}_{2^8}^{16}$, and c^r are vector constants dependent only on S. Note the shared application of γ.

The complete cipher Anubis initializes the state as $\sigma[K^0]$ applied to the input. This gets iteratively transformed through 12 rounds by $\sigma[K^r] \circ \theta \circ \tau \circ \gamma$ where the last round omits θ.

2.2 The Khazad Cipher

The Khazad block cipher [3] works on 8-byte blocks and uses a 16-byte key. The state is viewed as an element of $\mathbb{F}_{2^8}^8$. The components are similar to those of Anubis in many respects; the nonlinear layer γ remains the same. A description of the other components follows.

The linear diffusion layer θ This linear layer multiplies the input vector by the symmetric matrix

$$H = \begin{bmatrix} 01 & 03 & 04 & 05 & 06 & 08 & 0B & 07 \\ 03 & 01 & 05 & 04 & 08 & 06 & 07 & 0B \\ 04 & 05 & 01 & 03 & 0B & 07 & 06 & 08 \\ 05 & 04 & 03 & 01 & 07 & 0B & 08 & 06 \\ 06 & 08 & 0B & 07 & 01 & 03 & 04 & 05 \\ 08 & 06 & 07 & 0B & 03 & 01 & 05 & 04 \\ 0B & 07 & 06 & 08 & 04 & 05 & 01 & 03 \\ 07 & 0B & 08 & 06 & 05 & 04 & 03 & 01 \end{bmatrix}$$

and $\theta : a \mapsto a \cdot H$.

The key schedule Round keys satisfy $K^r = (\sigma[K^{r-2}] \circ \sigma[c^r] \circ \theta \circ \gamma)(K^{r-1})$ where $0 \leq r \leq 8$ and K^{-2} and K^{-1} are the first and second eight bytes of the key K, respectively. There is no component corresponding to the key extraction ω in Anubis.

The complete cipher Khazad initializes the state as $\sigma[K^0]$ applied to the input. This gets iteratively transformed through eight rounds by $\sigma[K^r] \circ \theta \circ \gamma$ where the last round omits θ.

3 Implementations

This section presents constant-time yet efficient implementations of both Anubis and Khazad. It begins with some background on SIMD vector operations, focusing on Intel processors. Then, for each cipher, a discussion on two implementation strategies appears. The first is more of a SIMD approach, running one instance in the Anubis case and two for Khazad. The second is a bit-slice approach running eight and sixteen instances, respectfully.

3.1 Vector Operations

In 64-bit mode, processors with Streaming SIMD Extensions 3 (SSE3) can operate on 16 128-bit SIMD registers `xmm0` through `xmm15`. SSE3 and predecessors contain a wealth of instructions for parallel computation amongst these registers. Cryptosystem implementations usually restrict to a smaller subset of these instructions dealing with integer values. Supplemental SSE3 (SSSE3) introduces a handful of new instructions, the most interesting for this work being a byte shuffler `pshufb`. Note that recent AMD processors implement SSE3 but not SSSE3, although a related instruction is slated for the eXtended Operations (XOP) extension.

Byte Shuffling Since the implementations in this work make heavy use of `pshufb`, a brief description of the instruction is in order. The name already implies the ability to shuffle bytes around in a vector, but perhaps hides an important aspect of the instruction. Aranha, López, and Hankerson note its versatility [1, Sect. 2.1]:

> "A powerful use of this instruction is to perform 16 simultaneous lookups in a 16-byte lookup table."

Formally, given 16-signed-byte vector operands a and b, components of the 16-byte vector output r of `pshufb` satisfy

$$r_i = \begin{cases} a_{b_i \bmod 16} & \text{if } b_i \geq 0, \\ 0 & \text{otherwise,} \end{cases}$$

so b holds the indices into the table and a the values. Indeed, this allows to implement any $\mathbb{F}_2^4 \to \mathbb{F}_2^8$ function in parallel: this is a constant-time hardware lookup table, shuffling the values in a based on the indices in b. To summarize, typical use of `pshufb` is either that of shuffling bytes around in a fixed manner (b is fixed) or implementing lookups into a fixed table (a is fixed), and the distinction is in the operand order.

Linear Maps Given the above, one can implement a linear map $\phi : \mathbb{F}_2^8 \to \mathbb{F}_2^8$ on 16 bytes in parallel. Denote $\alpha \in \mathbb{F}_{2^8}$ by $\alpha = \alpha_H x^4 + \alpha_L$ where α_i are the 4-bit nibbles. Linearity ensures $\phi(\alpha) = \phi(\alpha_H x^4) + \phi(\alpha_L)$ and each input on the right is effectively only four bits. Denote 16-byte vectors $t_{\phi H}$ and $t_{\phi L}$ that map the corresponding input to the output; these are the a from the previous section. The following steps realize ϕ in parallel:

1. Mask the lower nibble (α_L) of each byte in the input vector. (`pand`)
2. Bit-shift the input four positions towards LSB and mask again (α_H). (`psrlq`, `pand`)
3. Shuffle $t_{\phi L}$ and $t_{\phi H}$ with their respective indices from the above steps. (`pshufb` $\times$ 2)

4. Bitwise XOR the two outputs together. (`pxor`)

The second mask is a minor inconvenience due to the lack of an instruction to shift bits of individual bytes in a 16-byte vector (there are no `psllb` and `psrlb` instructions). The following implementations uses this strategy often. Note that when applying multiple maps to the same input, the first two steps are needed only once.

3.2 Implementing Anubis

This following presents SSSE3 implementation techniques for Anubis; it discusses two different approaches. The first is serial in the sense that it employs only one cipher instance, while the second runs eight instances in parallel.

A SIMD Approach Beginning with the nonlinear layer γ, the authors state that the choice to build it as an SPN with mini-boxes was influenced by efficient hardware implementation [2, 6.2]. A key observation in this work is that as a consequence of the underlying smaller $\mathbb{F}_{2^4} \to \mathbb{F}_{2^4}$ mini-boxes, the composition can be implemented elegantly using `pshufb`. Since P and Q are four bits to four bits but the instruction allows a parallel four bit to eight bit lookup, the bit permutations following P and Q can be unrolled for each layer to provide shifted and spread versions of their output. For example, $Q(\texttt{0x1}) = \texttt{0xE} = \texttt{1110}_2$ but following the first layer the upper two bits get shifted two positions towards the MSB: here the lookup provides $Q_0(\texttt{0x1}) = \texttt{0x32} = \texttt{110010}_2$. This unrolling yields the following six lookup tables for the corresponding layers:

$$
\begin{aligned}
t_{Q0} &= \texttt{0x20012313311000333003022212113221} \\
t_{P0} &= \texttt{0x0408804C488488C4C08C404400C8CC0C} \\
t_{P1} &= \texttt{0x01022013122122313023101100323303} \\
t_{Q1} &= \texttt{0x80048C4CC44000CCC00C08884844C884} \\
t_{Q2} &= \texttt{0x08010B070D04000F0C03020A06050E09} \\
t_{P2} &= \texttt{0x102080706090A0D0C0B0405000E0F030}.
\end{aligned}
$$

As the last layer does not permute the bits, note t_{P2} and t_{Q2} are simply the nibble-shifted and original contents as bytes, respectively, of P and Q.

With these tables in hand, the following steps implement layer i of S:

1. Mask the lower nibble of each byte in the input vector. (`pand`)
2. Bit-shift the input four positions towards LSB and mask again. (`psrlq`, `pand`)
3. Shuffle t_{Pi} and t_{Qi} with their respective indices from the above steps. (`pshufb` $\times$ 2)
4. Bitwise OR the two outputs together. (`por`)

Iterating this concept for all layers shows that S can be realized in parallel on all 16 input bytes using six `pand`, three `psrlq`, six `pshufb`, and three `por`.

Another option is to pair-wise reverse the wires on one mini-box per layer and use an XOR swap on two bits instead to implement the permutations between layers. It seems this does not reduce the operation count for current Intel processors.

Moving on to other components, the naïve way to implement the transpose τ requires a single `pshufb` instruction with indices defined as

$$t_\tau = \texttt{0x0F0B07030E0A06020D0905010C080400}$$

but in fact, by modifying the cipher and key schedule appropriately τ can be omitted. Consider the operation of rounds 1 and 2:

$$\sigma[K^2] \circ \theta \circ \tau \circ \gamma \circ \sigma[K^1] \circ \theta \circ \tau \circ \gamma.$$

With H a symmetric matrix ($H^T = H$), observe that the composition $\theta \circ \tau$ yields $a^T \cdot H = (H \cdot a)^T$. Denote $\hat{K}^1$ as τ applied to K^1 and $\hat{\theta} : a \mapsto H \cdot a$. Note γ is invariant under τ; it is not affected by any byte ordering. Then the following expression, essentially relying on the fact that τ is an involution, yields the same output:

$$\sigma[K^2] \circ \theta \circ \gamma \circ \sigma[\hat{K}^1] \circ \hat{\theta} \circ \gamma.$$

Hence all even rounds use the unmodified round keys and θ while odd rounds use transposed round keys and $\hat{\theta}$. With an even number of rounds, τ never needs to be applied during cipher operation. This is similar in spirit to Hamburg eliminating ShiftRows when implementing AES [10, 4.2].

For the linear layers θ and $\hat{\theta}$, viewing the input vector components as $a_i \in \mathbb{F}_{2^8}$, examining the matrix products reveals we need $a_i b_j$ for all i and all $b_j \in \{1, x, x^2, x^2+x\}$. That is, we need the result of three distinct linear maps applied to the input. Applying the machinery from Sect. 3.1 yields $t_2 = ax$ and $t_4 = ax^2$, then the final product is $t_6 = t_4 + t_2$. The outputs of θ and $\hat{\theta}$ differ only in how these t_i are subsequently shuffled. For θ, these vectors are shuffled using the following indices corresponding to their positions in the columns of H:

$$t_{\theta 2} = \texttt{0x0E0F0C0D0A0B08090607040502030001}$$
$$t_{\theta 4} = \texttt{0x0D0C0F0E09080B0A0504070601000302}$$
$$t_{\theta 6} = \texttt{0x0C0D0E0F08090A0B0405060700010203}$$

and the output is the XOR-sum of these three shuffled vectors with the input. This strategy realizes θ using seven `pshufb`, six `pxor`, two `pand`, and one `psrlq`. Note the `pand` can be eliminated by merging these layers with γ; the last layer of S does not permute the bits so the output from the final P and Q can be used directly as the indices for the linear maps. The byte shuffles for $\hat{\theta}$ are much more regular; for example

$$t_{\hat{\theta}2} = \texttt{0x0B0A09080F0E0D0C0302010007060504}$$

which in fact is not a byte shuffle but a dword shuffle `pshufd` that is more efficient since it takes an immediate operand.

For the key schedule, it remains to implement both the permutation π and key extraction ω; the former requires only one `pshufb` instruction with indices defined as

$$t_\pi = \texttt{0x0306090C0F0205080B0E0104070A0D00}.$$

Unfortunately ω is quite a different situation compared to θ, where the product of every entry in the matrix with every component of the input vector a is required. For example, here $(x^2 + x)a_i$ is only needed for $4 \leq i < 8$. When computing with 16-component vectors, this kind of selective computation is difficult to accomplish in an elegant fashion.

On the other hand, realizing multiple linear maps as in Sect. 3.1 with the same input amortizes the cost of the first two steps: the nibbles (indices into tables) need be produced only once. In light of this, one strategy is over-computation by producing $a_i b_j$ for all i and all b_j as distinct entries in V. Computing six of the maps (`02`, `04`, `08`, `14`, `3A`, and `40`) is enough to reach the remaining two with XOR chains (`06` and `78`). This strategy uses twelve `pshufb`, nine `pxor`, two `pand`, and one `psrlq`.

Denote the resulting vectors by r_i; these need to be combined at different indices before XOR-summing them to arrive at the result (three `pxor`). For column j of V with entries $[v_{0j}, v_{1j}, v_{2j}, v_{3j}]$ the needed vector is

$$[v_{0j}[a_{4j}, \ldots, a_{4j+3}], v_{1j}[a_{4j}, \ldots, a_{4j+3}], v_{2j}[a_{4j}, \ldots, a_{4j+3}], v_{3j}[a_{4j}, \ldots, a_{4j+3}]].$$

One way to achieve this is through a series of interleaves: `punpckldq` interleaves the lower two 4-byte values in the first operand with those in the second, and `punpckhqdq` the high 8-byte value.

The following illustrates this concept with $j = 1$ where vectors $\{r_1 = a, r_2 = ax, r_6 = a(x^2 + x), r_8 = ax^3\}$ facilitate constructing the vector

$$[a_4, a_5, a_6, a_7, \texttt{02}a_4, \texttt{02}a_5, \texttt{02}a_6, \texttt{02}a_7, \texttt{06}a_4, \texttt{06}a_5, \texttt{06}a_6, \texttt{06}a_7, \texttt{08}a_4, \texttt{08}a_5, \texttt{08}a_6, \texttt{08}a_7].$$

Here the r_i are filled with dummy data to help observe the interleaving action:

$$\begin{aligned}
r_1 &= \texttt{0x33333333222222221111111100000000} \\
r_2 &= \texttt{0x77777777666666665555555544444444} \\
r_6 &= \texttt{0xBBBBBBBBAAAAAAAA9999999988888888} \\
r_8 &= \texttt{0xFFFFFFFFEEEEEEEEDDDDDDDDCCCCCCCC} \\
t_0 &= \texttt{0x5555555511111111444444440000000}\texttt{0} \ \texttt{(punpckldq)} \\
t_1 &= \texttt{0xDDDDDDDD99999999CCCCCCCC88888888} \ \texttt{(punpckldq)} \\
t_2 &= \texttt{0xDDDDDDDD99999999555555551111111}\texttt{1} \ \texttt{(punpckhqdq)}.
\end{aligned}$$

These operations accomplish the goal of extracting bytes $v_4, \ldots, v_7$ from each of the given $v = r_i$ to a vector in a specific order corresponding to column j of V. The vectors for other j are obtained similarly with three instructions, but different interleaves. The exception being $j = 0$, using only one `pshufd` to broadcast the lower 4-byte value of the input across the vector.

A Bit-slice Approach Käsper and Schwabe use the SIMD registers to represent eight AES instances running in parallel [11]. While these can be unrelated instances with different keys, parallel block cipher modes such as counter mode are where this method is particularly interesting: encrypting the next eight counter values under one key in parallel. Eight SIMD registers hold the entire state for these eight instances, but each register represents one bit-slice of the state bytes for all instances.

Naturally, the same approach can be used to implement Anubis in counter mode. Denote 128-bit SIMD registers r_i for $0 \leq i < 7$ each holding bit i of all state bytes. Byte j of r_i holds bit i of the jth state byte for all eight instances, each instance at a fixed offset within these bytes. Figure 2 depicts this structure.

With this representation, some of the components from the previous section remain unchanged and are simply iterated for each r_i. For example, τ, π, and the shuffles at the end of θ. As this counter mode implementation uses only a single key, the key schedule components stay the same, but the resulting round keys must be subsequently converted into bit-slice format using eight times the storage. See [11, Sect. 4.1] for a brief discussion on general data conversion to and from bit-slice format. This implementation uses the same code for said conversion.

The two components that differ significantly in implementation compared to the serial case are the nonlinear layer γ and linear layer θ ($\hat{\theta}$), the only layers where any time consuming operations are carried out during encryption. The previous serial implementation relies heavily on `pshufb` as a lookup table to realize γ. In contrast, bit-slicing relies on boolean expressions alone to evaluate the S-box, facilitated by access to individual bits of all state bytes collected in one register. Indeed, this is the allure of bit-slicing.

The specification gives boolean expressions for P and Q with 18 gates each, implementing S with 108 gates [2, Appx. B]. This is not significantly lighter than the current smallest published AES S-box with 115 gates [6], although the former appeared at inception while the later took roughly a decade of research to whittle down, and further they are not immediately comparable as the later employs XNOR gates. Regardless, in software register-to-register moves must also be considered since most SSE instructions, particularly those for bitwise operations, do not allow passing a separate destination operand. The simple construction of S as an SPN using smaller P and Q easily allows the implementation to remain entirely within the working register set: the stack is not required, and in this work the implementation of γ uses 148 instructions. Table 1 compares the

	byte 15								⋯	byte 1								byte 0							
bit 0 (`xmm0`)	instance 7	instance 6	instance 5	instance 4	instance 3	instance 2	instance 1	instance 0	⋯	instance 7	instance 6	instance 5	instance 4	instance 3	instance 2	instance 1	instance 0	instance 7	instance 6	instance 5	instance 4	instance 3	instance 2	instance 1	instance 0
bit 1 (`xmm1`)																									
⋯																									
bit 7 (`xmm7`)									⋯																

Fig. 2. Bit-slice state representation for Anubis.

instruction counts to that of AES [11, Tbl. 2] and the result suggests, when bitslicing in software, the Anubis S-box is slightly more efficient compared to that of AES. In practice, instruction scheduling is equally important: alas a succinct, meaningful comparison is not straightforward.

For the linear layer, similar to an AES MixColumns, viewing the input and output of θ as matrices one can derive a formula for each byte of the output:

$$b_{ij} = a_{ij} + x(a_{i1-j} + a_{i3-j} + x(a_{i2+j} + a_{i3-j}))$$

where all the subscripts are modulo 4. Each multiplication by x implies three XOR gates for reduction. This leads to a cost of 38 `pxor` and 24 `pshufb` (`pshufd` for $\hat{\theta}$), notably heavier than the 27 `pxor` and 16 `pshufd` of MixColumns [11, Sect. 4.4]. The difference in `pxor` counts is simply due to the fact that the entries of H have higher degree than those for MixColumns, and the above formula for each byte contains one extra term in the sum. The difference in shuffle counts is due to the fact that the shuffles for MixColumns are simple dword rotations, and one can reduce the required shuffles per bit from three to two. The shuffles for H are not as simple and do not seem to allow this.

3.3 Implementing Khazad

This section presents two Khazad implementations, analogous to the previous two Anubis implementations. Both require a parallel block cipher mode when only a single key is used. The strategies are in fact so similar to those of Anubis that only a brief summary is provided. The implementations of the nonlinear layer γ stay the same; the key extraction ω and permutations π and τ in Anubis have no equivalent in Khazad, so the only component to consider is the linear layer θ.

Two Parallel Instances As the SIMD registers are 16-byte and Khazad maintains an 8-byte state, here the analogous SIMD implementation of Khazad runs two instances in parallel, for convenience restricted here to the same key using counter mode. The strategy to compute θ is the same as the corresponding layer in Anubis. First compute three linear maps (`02`, `04`, and `08`) and derive the remaining maps with XOR chains. The output is the XOR-sum of the input and the seven shuffled vectors resulting from the linear maps. This implementation uses 15 `pxor`, 13 `pshufb`, two `pand` and one `psrlq`.

Table 1. S-box instruction counts compared.

	`pxor`	`pand/por`	`movdqa`	Total
AES	93	35	35	163
Anubis	66	42	39	147

Sixteen Parallel Instances Lastly, the bit-slice implementation of Khazad in counter mode. Khazad works on 8-byte blocks and with 128-bit SIMD registers aligning the bits of bytes in the state, this implies 16 parallel streams. The approach to implement θ is exactly the same as with the bit-slice Anubis implementation: derive a formula for the output bytes and accumulate the result in output bits iteratively. For each of the eight input bits, this works out to 14 `pxor` and seven `pshufb` to produce a degree-10 polynomial. Similarly the reduction uses a total of 12 `pxor` to clear the three top bits.

4 Results

This section presents the timing results for all of the implementations described in this paper. The machine used for benchmarking is an Intel Core 2 Duo E8400 "Wolfdale" (45 nm) with 4GB of memory running Ubuntu 9.10, kernel 2.6.31-21, and gcc 4.4.1. Table 2 contains the timings for long streams. Timings are median over 1K runs obtained from the CPU time stamp counter `rdtsc`.

To place the results in some context, benchmark results of existing AES code running on the same machine are included as well. Hamburg's AES implementations includes a benchmarking script and the reported time is for encrypting 4kB [10]. Käsper and Schwabe implement the eSTREAM API that benchmarks a number of different metrics; the reported time is the best result from the test suite, that of "Encrypted 60 packets of 1500 bytes (under 1 keys, 60 packets/key)".

Note that one purpose of this work is to improve the security and, if possible, speed of Anubis and Khazad software implementations. Hence the AES timings are only included as a rough benchmark and are not for direct comparison. In particular, AES-128 has 10 rounds while Anubis-128 has 12. They have very different code footprints: AES encryption and decryption are implemented separately, while with Anubis and Khazad they only differ in the key schedule.

The timings in Table 2 show that the serial Anubis implementations outline here are very competitive with the purely table-based implementation. In particular, there is no significant penalty to realize protection against cache-timing attacks on this platform. The compiler is able to optimize the C implementation using compiler intrinsics for SIMD operations quite well; it is unclear how to improve it by hand-crafted assembly. For parallel modes, the bit-slice approach is significantly more efficient than the serial approach for both Anubis and Khazad.

5 Conclusion

This paper presents a number of constant-time implementations of the Anubis and Khazad block ciphers. The results show that constant-time and efficient are not mutually exclusive with respect to their software implementation. The work here also further showcases the potential of a vector-byte shuffle instruction to provide both secure and fast software implementations of cryptosystems.

Table 2. Timing results in cycles per byte.

Cipher	Method	Language	Mode	Instances	"Wolfdale"
Anubis	SSSE3	C	CTR	1	21.7
Anubis	SSSE3	C	CBC	1	20.7
Anubis	SSSE3	C	CBC^{-1}	1	20.3
Anubis	SSSE3	asm	CTR	8	9.2
Anubis	Table	C [2]	CTR	1	20.7
Anubis	Table	C [2]	CBC	1	21.3
Anubis	Table	C [2]	CBC^{-1}	1	21.2
Khazad	SSSE3	asm	CTR	2	18.6
Khazad	SSSE3	asm	CTR	16	10.3
Khazad	Table	C [3]	CTR	1	19.8
AES	SSSE3	asm [10]	CTR	1	11.6
AES	SSSE3	asm [10]	CBC	1	11.0
AES	SSSE3	asm [10]	CBC^{-1}	1	13.6
AES	SSSE3	asm [11]	CTR	8	8.0

It is worth mentioning that at least two other primitives make use of the compact S-box used in Anubis and Khazad [4, 15]. Its particularly efficient software implementation here, in serial using `pshufb` or parallel when bit-slicing, greatly encourages further use: perhaps as a building block for other primitives.

Realizing the threat that timing attacks pose to software implementations, more recent trends in cipher design are away from the rather traditional view of an S-box as a lookup table towards methods that better suit constant-time implementations using native instructions supported by common processors. For example, the Threefish block cipher explicitly states this as a design criteria [9, Sect. 8.1], using an extremely simple nonlinear function MIX consisting of a rotation, XOR, and addition modulo 2^{64} iterated during a large number of rounds. However, equipped with powerful instructions like `pshufb` it will be interesting to see how cryptologists harness this machinery and what the future holds for cipher design.

References

1. Aranha, D.F., López, J., Hankerson, D.: High-speed parallel software implementation of the η_T pairing. In: Pieprzyk, J. (ed.) CT-RSA. Lecture Notes in Computer Science, vol. 5985, pp. 89–105. Springer (2010)
2. Barreto, P.S.L.M., Rijmen, V.: The Anubis block cipher. `http://www.larc.usp.br/~pbarreto/anubis-tweak.zip` (2001)
3. Barreto, P.S.L.M., Rijmen, V.: The Khazad legacy-level block cipher. `http://www.larc.usp.br/~pbarreto/khazad-tweak.zip` (2001)
4. Barreto, P.S.L.M., Simplício Jr., M.A.: CURUPIRA, a block cipher for constrained platforms. In: 25th Brazilian Symposium on Computer Networks and Distributed Systems. pp. 61–74 (2007), `http://www.sbrc2007.ufpa.br/anais/2007/ST02%20-%2001.pdf`

5. Bernstein, D.J.: Cache-timing attacks on AES. http://cr.yp.to/papers.html#cachetiming (2004)
6. Boyar, J., Peralta, R.: New logic minimization techniques with applications to cryptology. Cryptology ePrint Archive, Report 2009/191 (2009), http://eprint.iacr.org/
7. Canright, D., Osvik, D.A.: A more compact AES. In: Jr., M.J.J., Rijmen, V., Safavi-Naini, R. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 5867, pp. 157–169. Springer (2009)
8. Clavier, C., Gaj, K. (eds.): Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings, Lecture Notes in Computer Science, vol. 5747. Springer (2009)
9. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to NIST (Round 2) (2009), http://www.skein-hash.info/sites/default/files/skein1.2.pdf
10. Hamburg, M.: Accelerating AES with vector permute instructions. In: Clavier and Gaj [8], pp. 18–32
11. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier and Gaj [8], pp. 1–17
12. Preneel, B.: The NESSIE project: towards new cryptographic algorithms. In: Information Security Applications, 3rd International Workshop, WISA 2002. pp. 16–33 (2002)
13. Rudra, A., Dubey, P.K., Jutla, C.S., Kumar, V., Rao, J.R., Rohatgi, P.: Efficient Rijndael encryption implementation with composite field arithmetic. In: Çetin Kaya Koç, Naccache, D., Paar, C. (eds.) CHES. Lecture Notes in Computer Science, vol. 2162, pp. 171–184. Springer (2001)
14. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with s-box optimization. In: Boyd, C. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 2248, pp. 239–254. Springer (2001)
15. Simplício Jr., M.A., Barreto, P.S.L.M., Carvalho, T.C.M.B., Margi, C.B., Näslund, M.: The CURUPIRA-2 block cipher for constrained platforms: Specification and benchmarking. In: Bettini, C., Jajodia, S., Samarati, P., Wang, X.S. (eds.) PiLBA. CEUR Workshop Proceedings, vol. 397. CEUR-WS.org (2008)