Fast Software Encryption with SIMD

How to speed up symmetric block ciphers with the AVX/AVX2 instruction set

Johannes Götzfried Tilo Müller
Department of Computer Science
Friedrich-Alexander University of Erlangen-Nuremberg
{johannes.goetzfried,tilo.mueller}@cs.fau.de

ABSTRACT

Symmetric block ciphers play an important role in today's IT security and are generally used by everybody who is working with electronic devices on a daily basis. We show a generic approach to speed up block ciphers with SIMD instructions that are provided on the latest x86 CPUs. With SIMD, processing multiple input blocks can be parallelized. To verify the effectiveness of our approach, we exemplarily provide implementations of five common block ciphers taken from the Crypto-API of the Linux kernel. We make use of the *Advanced Vector Extensions* (AVX), a novel instruction set that was released by Intel in 2011, and its successor AVX2, which will become available on mass-market CPUs in 2013. We give a detailed performance analysis of the AVX variants, and an estimation for the performance of the AVX2 variants. Each of our implementations outperforms the previously fastest one. We submitted the AVX variants to the Linux kernel and four of them were already merged into mainline.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; E.3 [**Data**]: Data Encryption

General Terms

Security

Keywords

Streaming SIMD Extensions, Advanced Vector Extensions, Symmetric Block Ciphers, Performance, Linux Kernel

1. INTRODUCTION

Today, encryption is used (often unconsciously) by everybody who uses the Internet on a daily basis, e.g., in terms of network communication protocols like HTTP/SSL, VPNs and WiFi that encrypt data against eavesdropping. Moreover, disk encryption is an increasingly popular method to protect nomadic laptops against physical loss and theft. Since more and more employees carry around confidential data on mobile devices, and work outside their company locations, they are often forced by company regulations to use disk and network encryption. In both cases, the performance of symmetric block ciphers plays a vital role for the user experience and work efficiency. Indeed, cryptography must necessarily involve a performance drawback, but it is an interesting and important research topic how this performance drawback can be minimized.

In practice, low-level implementations in assembly language can oftentimes outperform the generic C implementation of a cipher, because high level languages and general purpose compilers do not support the latest CPU features well. This is particularly the case for *Single Instruction Multiple Data* (SIMD) instruction sets. SIMD instructions enable programmers to process multiple data with only one instruction, thereby saving CPU clocks. SIMD instructions were first introduced by Intel as MMX (*Multi Media eXtension*) in 1997, and later as SSE (*Streaming SIMD Extensions*) in 2001. The state-of-the-art SIMD instruction set is AVX (*Advanced Vector Extensions*) which is available since 2011. Its successor AVX2 will become available in 2013.

1.1 Contributions

In this paper, we make the following contributions:

- We introduce a generic approach to speed up symmetric block ciphers by means of modern SIMD instructions. Our approach is based on the concept to process sequenced input blocks of a cipher in parallel. Therefore, our approach is particularly efficient in combination with modes of operation like CBC and ECB. The encryption of single blocks is not (or only slightly) improved.
- To prove the effectiveness of our approach, we developed AVX variants for five different ciphers taken from the Linux Crypto-API (Serpent [3], Twofish [16], Blowfish [15], Cast-128 [1], and Cast-256 [2]). We provide our implementations as open source kernel patches. We also submitted four of them to the official kernel, and they were accepted and merged into mainline. Integrating our ciphers into the Linux kernel makes them available for a wide range of uses, because existing programs that rely on the Crypto-API can directly benefit from them (e.g., *dm-crypt* for disk encryption).
- Since AVX lacks instructions that are necessary to fully parallelize block ciphers, we additionally implemented

EUROSEC '13, April 14, 2013 Prague, Czech Republic

each cipher with the upcoming instruction set AVX2. These implementations were developed on an emulator, and will first run on CPUs available on the market in 2013. All implementations are available under an open source license at `http://www1.cs.fau.de/avx.crypto`.

- We provide performance benchmarks for our AVX implementations in user and in kernel mode, and for the case of disk encryption. In practice, we achieved performance speedups of up to 115% compared to the previously fastest C-implementation in the Linux kernel, and up to 30% compared to the fastest assembler implementation. Moreover, we estimate that our AVX2 implementations are considerably faster than the AVX variants.

The main performance advantage of our implementation stems from the fact that we parallelize the processing of input blocks for different modes of operation.

1.2 Related Work

There are AVX-based implementations of hash primitives, in particular of the SHA-3 finalists Blake [14] and Grøstl [10]. However, we could not find any such implementation for a block cipher; during the time of this writing, no AVX-based implementation of a symmetric block cipher is known (besides ours). Nevertheless, block ciphers were frequently considered for high-performance implementations in the past.

In 2012, for example, Gilger, Barnickel and Meyer [6], demonstrated how to exploit graphic processing units (GPUs) to speed up symmetric block ciphers. And in 2008, Intel proposed an instruction set called AESNI [7] which implements the AES cipher [12] efficiently in hardware. AESNI is known to cause outstanding performance benchmarks on CPUs where this instruction set is available (i.e., on Intel's Core-i5 and Core-i7 series). A special instruction set to speed up the performance of a block cipher, however, can only be expected to be introduced for standards like AES. All remaining ciphers, like Serpent and Twofish, can not benefit from specialized instruction sets but must be implemented with general purpose instructions such as SIMD.

1.3 Paper Outline

The remainder of this paper is structured as follows: In Sect. 2, we provide background information about symmetric block ciphers and SIMD instructions. In Sect. 3, we describe design choices and technical details behind our implementations. In Sect. 4, we present performance benchmarks for AVX, and an estimation of the performance for AVX2. Finally, in Sect. 5, we conclude with a suggestion of future research directions.

2. BACKGROUND

We now provide necessary information about symmetric block ciphers (Sect. 2.1), as well as about SIMD and AVX (Sect. 2.2).

2.1 Symmetric Block Ciphers

Symmetric block ciphers use a single *secret key*, which is typically between 64 and 512 bits, to encrypt a plaintext and to decrypt the corresponding ciphertext. Unencrypted input blocks, which are typically of a fixed length between 64 and 128 bits, are scrambled to encrypted output blocks, which are typically of the same length than the input block. Additionally, all modern ciphers essentially have a similar structure. They get initialized with the secret key and generate a set of *round keys* from the key bits (a.k.a. the *key schedule*). In the encryption and decryption routines, the data is then processed by means of identical rounds. In each round, a different round key is applied. Common operations of these rounds are substitutions, permutations and key mixing (e.g., with *xor*). In modern block ciphers, these operations are not performed on byte level but on larger parts for performance reasons (often on doublewords, i.e., on 32-bit level). As we will show, this fact can be well exploited for parallelization.

To process input messages of arbitrary length, symmetric block ciphers are used in combination with *modes of operation* [11] [13]. Modes of operation combine the encryption of single input blocks in a secure manner. We implemented not only the block ciphers, but also the modes of operation (in particular ECB, CBC, CTR, LRW and XTS).

2.2 Advanced Vector Extensions

All basic scalar instructions like *add* and *xor* have SIMD equivalents which can be used to operate on different values in parallel. Common SIMD instructions operate on large registers that are, for example, 128 or 256 bits wide. These instructions can modify separated parts of a register in a similar fashion at once. For example, instead of adding two data sets (x_1, y_1) and (x_2, y_2) sequentially (by first computing $x_1 + x_2$ and then $y_1 + y_2$), SIMD can add both values with a single instruction. As a consequence, many instructions can be saved by SIMD provided that an algorithm is suitable for being parallelized.

AVX [9] extends earlier SIMD instructions like MMX and SSE with a nondestructive, three-operand syntax. Moreover, with AVX, sixteen SIMD registers denoted to as YMM0 to YMM15 have been introduced, each comprising 256 bits. The lower 128 bits of YMM registers correspond to respective 128-bit wide registers from SSE, denoted to as XMM0 to XMM15. The data stored in a YMM register can be interpreted in different ways, depending on the instruction that operates on the register. For example, the data can be interpreted as "packed doublewords", meaning as eight 32-bit chunks. Each 32-bit chunk is then processed separately.

One drawback with AVX is that YMM registers as a whole can only be used in combination with floating point types. Such a design is meaningful in terms of multimedia applications, but lacks in support for cryptography. Symmetric block ciphers are never based on floating point operations but on packed integer values. That is why AVX-based cipher implementations cannot benefit well from entire 256-bit registers. Nevertheless, we can benefit from their separation into "double quadwords" (i.e., into 128-bit chunks), interpreted as packed integers. Moreover, we can benefit from the three-operand syntax of AVX.

AVX2 eliminates the drawback of AVX because it supports packed integer operations on whole 256-bit registers (YMM0 to YMM15). Another advantage of AVX2 are the *gather* instructions. These instructions allow to conditionally gather packed doublewords (32-bit) from memory using signed doublewords as indices, and to merge these values into a destination register. A *gather* operation is basically a scale-index-base addressing operation known from general purpose instructions, but in addition *many indices* can be defined, and many values can be looked up at once. Such

operations are useful to parallelize table lookups. The first processor generation supporting AVX2 will be the Haswell architecture whose market launch is expected in 2013 [8].

3. IMPLEMENTATION

We now present the design and implementation of the block ciphers Serpent, Twofish, Blowfish, Cast-128, and Cast-256. In Sect. 3.1, we present our AVX implementations, and in Sect. 3.2, we present our AVX2 implementations. We can not give a detailed description for each cipher, but we rather focus on describing approaches to parallelize symmetric block ciphers in general. In Sect. 3.3, we introduce our Linux kernel patches, including the modes of operation.

3.1 AVX Implementation

We speed up the encryption and decryption routines of a cipher, but we do not modify the key scheduling part. The reason is that key scheduling is only executed once and is therefore not critical for the performance. Moreover, the key schedule can not (or only slightly) be optimized with our technique, because we speed up encryption by parallelizing sequenced input blocks. This technique has no meaningful counterpart in key scheduling. Encryption and decryption routines are usually executed on several input blocks at once, and can therefore be well optimized through parallelization.

In the remainder of this section, we only have a look to the case of *encryption*. In symmetric ciphers, the case of *decryption* is very similar to the case of encryption because decryption operations are usually applied in an inverse order. Moreover, we assume that the block size is 128 bits. Actually, the ciphers that we implemented have a block size of either 64-bit or 128-bit, but our generic approach does not change when dealing with a different block size.

Our implementation starts with fetching input blocks from main memory. We always process eight input blocks at one go. These input blocks are interpreted as two 4-block chunks, i.e., we actually parallelize the processing of four blocks. However, as it is generally a good idea to fetch as many blocks as possible into SIMD registers at one go (in order to save the repeated loading of round keys), we read eight blocks from memory into eight XMM registers of the CPU. One block exactly fits into one XMM register because both are 128 bits wide. We then apply a series of *unpack* operations on four sequenced registers, which together give a 4x4 matrix transposition of doublewords. Unpack operations do interleaving of doublewords and doublequadwords, and thus with reasoned combinations of unpack operations, a matrix transposition can be achieved. After the unpack operations, the first XMM register contains the first doubleword of four sequenced input blocks, the second XMM register contains the second doubleword of the four sequenced input blocks, and so forth.

In this initial step, we can save *move* operations by exploiting the three-operand syntax of AVX. Using a common two-operand syntax, we would have to save the destination registers of the unpack operations because they must be equal to one of the source registers. But with the AVX instructions, we can skip this extra *move* because the destination register can be different to the source registers.

After we loaded the input blocks into CPU registers, we can now perform the actual cipher routines for encryption. Symmetric block ciphers consist of basic arithmetic and logical operations that are applied on doublewords, like *xor*, *add*, *rotate*, and *shift*. The principle of parallelizing these operations is as follows: We replace each arithmetic and logical operation from general purpose instructions with an equivalent from SIMD. Basically, each arithmetic and logical operation can be replaced with a counterpart from SIMD. After doing so, a single assembler operation processes four doublewords of four different blocks in parallel, with only one packed SIMD instruction instead of processing doublewords one after another.

With this technique, we can reduce the number of arithmetic and logical operations by 75%. But we can do even better. Since eight input blocks are fetched into CPU registers at one go, we do not need to reload round keys for every single block, but need to do so only for every eighth block. By this, we additionally save 87.5% of the *move* instructions. Note that *move* instructions are costly as they involve memory.

In addition to the simple arithmetic and logical operations, symmetric block cipher have complex algebraic operations. For example, many block ciphers perform multiplications in the finite field $GF(2^8)$. These operations cannot be replaced by simple SIMD instructions. Therefore, we replace these operations by table lookups based on precomputed data sets. With AVX, however, we cannot parallelize table lookups. Instead, we have to extract single values from SIMD registers, move them into general purpose registers, do the table lookups, and finally reinsert them into SIMD registers. Of course, this is a time consuming task and has a serious impact on the performance of our implementation. As it turns out, however, the advantages explained above compensate this fact.

After modifying the actual cipher, as outlined above, we need to reverse the initial transformations. Since matrix transpositions are self-inverse, we can apply the same transformations as above. Afterwards, the eight sequenced blocks are written back to memory and then we are basically done. Nevertheless, implementing symmetric block ciphers with SIMD instructions also involves the modes of operation. Without an appropriate mode of operation, we cannot process more than one block in parallel. We come back to this point in Sect. 3.3.

3.2 AVX2 Implementation

We developed our AVX2 implementations in an emulator because CPUs supporting this instruction set are not available. With AVX2, we are (or will be) able to use whole 256-bit wide YMM registers. Therefore, we can process more blocks in parallel than with AVX, and consequently we can save even more instructions. Furthermore, with AVX2 we are able to overcome the performance bottleneck of table lookups.

Instead of eight input blocks, we can now fetch sixteen input blocks from memory at one go. This becomes possible since AVX2 enables us to use integer operations on whole 256-bit wide YMM registers. We first read the input blocks sequentially from memory into CPU registers, and we then apply the same input transformation as with AVX (i.e., the 4x4 matrix transposition; see Sect. 3.1).

Note that, instead of applying this transformation to four 256-bit YMM registers, we apply it to the four upper 128-bit halves and the four lower 128-bit halves separately. Besides this, the input transformation works with almost the same instructions as above. After the transformation, YMM0 holds

the first doublewords of eight sequenced input blocks, YMM1 holds the second doublewords of eight sequenced input blocks, and so forth. This means that we basically have the same situation as we had with AVX. In contrast to that, we just fetched sixteen input blocks at one go (instead of eight), and we process eight input blocks in parallel (instead of four). That is, we reduce the number of required arithmetic and logical operations by 87.5%, and the number of *move* instructions for round keys by 93.75%.

After transforming the input, we perform the actual cipher routines by means of AVX2. The arithmetic and logical operations, that we already replaced by AVX equivalents in Sect. 3.1, can now be replaced by AVX2 equivalents that work on entire 256-bit YMM registers. As these operations are packed operations on doublewords, there is no need for further transformations. The transformations that we applied initially suffice. An operation now processes eight doublewords of eight different blocks with a single packed SIMD instruction, instead of processing just one doubleword of one block.

In the AVX variant, we dealt with complex algebraic operations by consulting precomputed lookup tables. However, we had to move data into general purpose registers to perform table lookups. This turned out to be a very costly task. With AVX2, we can use the *gather* operation instead. By using the *gather* operation, we perform eight lookups of doublewords in parallel, without the need to move data into general purpose registers. Our lookup tables have the format 8×32, meaning that the index is 8-bit (byte) and the result is 32-bit (doubleword). To prepare the data in SIMD registers in a way that is suitable for byte indices, we apply packed logical right shifts and respective bitmasks. The important point is that, although we need instructions to prepare the indices, the data never leaves the SIMD register. In other words, we do not have to move any data into general purpose registers. We do not exactly know how long a *gather* operation takes on upcoming CPUs, but alone the fact that we save several *move* instructions between SIMD and general purpose registers, lets us believe that AVX2 implementations considerably outperform the AVX variants.

At the end, we need to reverse the initial transformations and write the output blocks back to memory. This works similar to the case of AVX, i.e., we can use the same matrix transposition as it is self-inverse.

To sum up, the advantages of AVX2 are twofold: First, we can process twice as many blocks with AVX2 than with AVX. Second, we can save many costly *move* instructions by means of the *gather* operation, because all parts of a cipher can now completely be parallelized. We conjecture that our AVX2 variants will show up a significant gain in performance.

3.3 Kernel Integration

To use our AVX implementations for disk encryption (and other Crypto-API based applications), we have to integrate them into the Linux kernel. To fully utilize parallelism in our implementations, we register our implementations together with the modes of operation to the Crypto-API. Inside the Crypto-API, we deal with modes of operation like CBC and ECB. Of course, that does only work if the mode of operation supports the parallel processing of sequenced blocks. This is the case for all five modes of operation that we implemented, except for the *encryption* routine of CBC (whereas *decryption* works fine for CBC). The reason for this stems from the structure of CBC which requires us to encrypt blocks sequentially. Without the result of the previous block, it is impossible to encrypt the next block. In Sect. 4, this fact becomes visible in the performance benchmarks.

We provide support for the modes of operation ECB, CBC, CTR, LRW, and XTS. For each mode of operation, a block cipher is registered to the Crypto-API of the kernel, e.g., *cbc(twofish)* or *ecb(serpent)*. The new block ciphers compete with older implementations of the same cipher if an algorithm is requested by name. To make use of the fast variants, new ciphers are assigned with a higher priority than the older ciphers. The Linux kernel then automatically chooses the best cipher module that is available.

For example, the generic C-implementation of Twofish registers itself to the Crypto-API with the name `twofish` and a priority of 100. If someone decides to use Twofish for disk encryption, the name `cbc(twofish)` is requested and the mode `cbc` is automatically combined with the cipher `twofish` by the Crypto-API. We provide both pieces in one registered entity and name it directly `cbc(twofish)` with a priority of 400. Now the Crypto-API chooses our implementation rather than the C implementation, provided that the user has loaded our module into the kernel.

For more technical details about our implementations, regarding both the AVX ciphers and the modes of operation, please refer to our open source Linux kernel patches. So far, we submitted kernel patches for the AVX implementations, but not for the AVX2 implementations. First, there is no support for AVX2 in the Linux kernel at the time of this writing, and second, CPUs with AVX2 are not available yet. However, as the differences between the integration of our AVX and AVX2 variants are minimal, we can quickly adapt patches once the hardware and the kernel are ready for them.

We provide our ciphers as loadable kernel modules and add entries for them in *Kconfig* (the build system of the Linux kernel), such that users can decide whether they want to make use of a new cipher or want to keep the old one. Compiling our ciphers directly into the Linux kernel (i.e., without LKM support) is possible as well.

4. PERFORMANCE BENCHMARKS

We now give performance benchmarks for our AVX and AVX2 implementations. In Sect. 4.1, we give details about the performance for AVX, and in Sect. 4.2, we give an estimation about the performance for AVX2. Since there are no CPUs with AVX2 available, concrete results must be limited to the case of AVX. For AVX, our tests were performed on an Intel Core i5-2500 CPU.

4.1 AVX Benchmarks

To give an overview of our results, we list for each AVX-based implementation the gain in performance compared to the previously fastest variant in the Linux kernel. Our AVX-based implementation of Serpent is 6% faster than the earlier SSE2-based implementation. Twofish is accelerated by 30% in comparison to the earlier assembly implementation, and Cast-128 is even by 115% faster than any previous implementation of it in the Linux kernel. For Cast-256, we get at least a speedup of 88%. For Blowfish, however, we were only able to achieve a speedup of 0.8%. Blowfish cannot be well parallelized with AVX because it mainly consists of table lookups rather than arithmetic and logical operations. Table lookups can only be parallelized with AVX2.

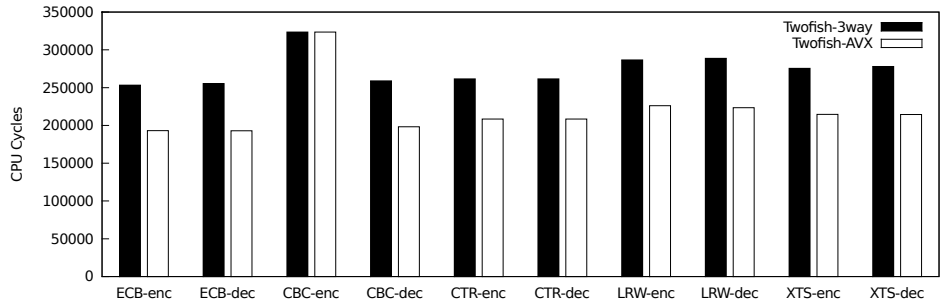

Figure 1: Twofish benchmarks for different modes of operation (256-bit key, 8192-byte input).

Implementation	Instructions	Time (s)	Speedup (%)
`generic`	35913728	6.215	-
`asm_64`	28788575	5.800	7.15
`asm_64-3way`	34493255	4.714	23.03
`avx`	28622848	3.605	30.79
`avx2`	6426624	-	-

Table 1: Twofish benchmarks in user mode.

Kernel Module	Disk Speed (MB/s)
`aes-x86_64`	318.68
`aesni-intel`	1055.75
`twofish-generic`	282.15
`twofish-x86_64`	314.98
`twofish-x86_64-3way`	390.15
`twofish-avx-x86_64`	467.49

Table 2: Disk encryption benchmarks for Twofish running in kernel mode (cbc-essiv:sha256).

We now exemplarily show the evaluation of Twofish in detail, because aside from AES, it is probably the most important symmetric block cipher today. Furthermore, Twofish is a good example to point out the differences between AVX and AVX2.

Table 1 lists benchmarks for five different Twofish implementations running in user mode. Each of these implementations is written in 64-bit assembler, except the generic implementation which is written in C. The instruction count is based on the encryption of one megabyte of data in ECB mode; timings were measured while encrypting one gigabyte of data. The speedup factors are based on the timing columns and show the gain in performance *in comparison to the previously listed implementation*. The first three implementations (*generic*, *asm_64*, and *asm_64-3way*) have already been present in the kernel before. Our AVX implementation is about 30% faster than the 3-way parallel implementation.

Figure 1 shows performance results for different modes of operation running in kernel mode. The key size is 256-bit and the input data comprises 8192 bytes. The speedup between the 3-way and the AVX implementation remains almost constant with all modes of operation, except for CBC/encryption.

Figure 2 visualizes the CBC decryption routine for two implementations of Twofish and AES. The AES implementations are shown as a reference value. The first graph shows the number of CPU cycles needed with increasing input size. The second graph shows the number of CPU cycles per byte with increasing input size (it has a logarithmic scale). The speedup of our AVX implementation in comparison to the 3-way variant is clearly visible. Of course, the AESNI-based implementation of AES beats all others but in the second graph it becomes visible that Twofish is a very fast cipher. The assembler implementation of Twofish always outperforms that of AES. For small data sizes, it even outperforms the AESNI-based implementation of AES. Furthermore, the graph visualizes that the Twofish implementations converge with increasing data sizes and stay proportional to each other. The absolute speed of our AVX implementation is about 24 cycles per byte.

Table 2 shows disk benchmarks of the Twofish implementations. To encrypt disks, we made use of Linux' *dm-crypt* interface. However, to get accurate results, the measurements were taken within a ramdisk because otherwise the results fluctuate. The ramdisk had a size of 2.5 GB. The speedup between the 3-way implementation and our AVX implementation has practical impact on applications using our cipher for disk encryption.

4.2 AVX2 Benchmarks

Continuing with the example of Twofish, we want to estimate the gain in performance that we can expect from our AVX2 variant. Since Twofish comprises many table lookups, we can not fully parallelize it with AVX. With AVX2, however, we could fully parallelize it as described above (Sect. 3.2). Hence, we expect a higher performance speedup with AVX2 than with AVX. Unfortunately, we cannot give concrete benchmarks because AVX2 will only be available on upcoming CPUs. Instead, we count the number of instructions.

Table 1 lists the number of necessary instructions for our AVX- and AVX2-based implementations for Twofish. Encrypting one megabyte of data, the AVX variant needs about 4.5 times more instructions than the AVX2 variant. The AVX2 implementation is consequently expected to be *much faster* than its AVX counterpart. Note that we got similar results for the other ciphers. We always experienced a huge decrease of the number of instructions, especially if the AVX implementation makes use of table lookups. Except for Serpent this has always been the case.

Nevertheless, our AVX2 variants must be considered as ongoing work, and we can hand in concrete performance results only in mid 2013.

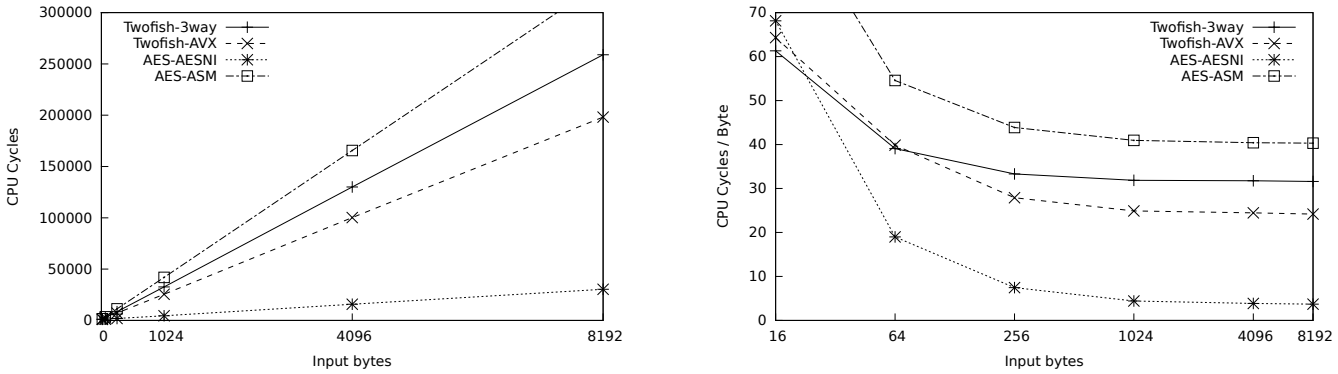

Figure 2: Twofish and AES benchmarks for CBC decryption (256-bit key).

5. CONCLUSIONS AND FUTURE WORK

Although computing power increases steadily, the performance of symmetric block ciphers remains an important topic in an increasingly mobile and networked world. There is a high demand for fast software encryption with symmetric block ciphers, e.g., for the case for disk encryption running in kernel mode, but also for user mode encryption like SSL. In this paper, we showed a generic approach to speed up symmetric block ciphers with SIMD instructions. To verify our approach, we presented fast implementations of the five block ciphers Serpent, Twofish, Blowfish, Cast-128 and Cast-256. With the choice of these algorithms, we have accelerated one of the most important symmetric block ciphers aside from AES. As our contribution, we implemented these ciphers with both AVX and AVX2, and we provided Linux kernel patches for four of them, which were already merged into Linux mainline.

Besides analyzing our AVX2 implementation on real hardware in the future, we want to provide AVX and AVX2 variants for more symmetric block ciphers, e.g., for Camellia [4]. Likewise, we want to provide AVX and AVX2 variants for hash algorithms (where that has not been done yet). Furthermore, another research direction is to port our ideas to other platforms, most notably to the ARM platform. Smartphones and tablet PCs are usually equipped with an ARM processor, and many of those run a Linux kernel (Android). Since disk encryption is an important issue for mobile devices, there is a high demand for fast software encryption on ARM. On ARM, however, there is no AVX or AVX2 instruction set available. The SIMD instruction set on ARM is called NEON [5], which is supported by the Cortex-A series and other ARM CPUs. Note that on ARM, there exists no AESNI instruction set like on modern x86 processors. This makes it even interesting to provide fast SIMD-based implementations of AES.

6. REFERENCES

[1] C. Adams. The CAST-128 Encryption Algorithm, May 1997. http://tools.ietf.org/html/rfc2144.

[2] C. Adams and J. Gilchrist. The CAST-256 Encryption Algorithm, June 1999. http://tools.ietf.org/html/rfc2612.

[3] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard, Mar. 2000. http://www.cl.cam.ac.uk/ rja14/Papers/serpent.pdf.

[4] K. Aoki, T. Ichikawa, and M. K. et. al. Specification of Camellia – a 128-bit Block Cipher, Sept. 2001. http://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf.

[5] ARM Ltd. NEON SIMD Instruction Set, June 2012. http://www.arm.com/products/processors/technologies/neon.php.

[6] J. Gilger, J. Barnickel, and U. Meyer. GPU-acceleration of block ciphers in the OpenSSL cryptographic library. In D. Gollmann and F. C. Freiling, editors, *Information Security 15th International Conference (ISC)*, Passau, Germany, Sept. 2012. Springer Berlin Heidelberg.

[7] Intel Corporation. *Intel Advanced Encryption Standard (AES) New Instructions Set*, 323641-001 edition, May 2010.

[8] Intel Corporation. Haswell New Instruction Descriptions, June 2011. http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/.

[9] Intel Corporation. *Intel Advanced Vector Extensions Programming Reference*, 319433-011 edition, June 2011.

[10] K. Matusiewicz, M. Schlaffer, and S. S. Thomsen. Grøstl Implementation Guide, Mar. 2012. http://www.groestl.info/groestl-implementation-guide.pdf.

[11] National Institute of Standards and Technology. *DES Modes of Operation*, FIPS PUB 81 edition, Dec. 1980.

[12] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*, FIPS PUB 197 edition, Nov. 2001.

[13] National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation*, special publication 800-38a edition, Dec. 2001.

[14] S. Neves and J.-P. Aumasson. Implementing BLAKE with AVX, AVX2, and XOP, Mar. 2012.

[15] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). *Cambridge Security Workshop Proceedings*, pages 191–204, 1993. http://www.schneier.com/paper-blowfish-fse.html.

[16] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Twofish: A 128-Bit Block Cipher, June 1998. http://www.schneier.com/paper-twofish-paper.pdf.