An Efficient Hash Function Construction for Sparse Data

Nir Soffer[1][a] and Erez Waisbard[2][b]
[1]*IBM, Givataim, Israel*
[2]*CyberArk, Petach Tikva, Israel*

Keywords: Integrity Verification, Hash Functions, Storage Virtualization, Sparse Disks, Parallel Computation.

Abstract: Verifying the integrity of files during transfer is a fundamental operation critical to ensuring data reliability and security. This is accomplished by computing and comparing a hash value generated from the file's contents by both the sender and the receiver. This process becomes prohibitively slow when dealing with large files, even in scenarios involving sparse disk images where significant portions of the file may be unallocated. We introduce *blkhash*, the first hash construction tailored specifically for optimizing hash computation performance in sparse disk images. Our approach addresses the inefficiencies inherent in traditional hashing algorithms by significantly reducing the computational overhead associated with unallocated areas within the file. Moreover, *blkhash* implements a parallel computation strategy that leverages multiple cores, further enhancing efficiency and scalability. We have implemented the *blkhash* construction and conducted extensive performance evaluations to assess its efficacy. Our results demonstrate remarkable improvements in hash computation speed, outperforming state-of-the-art hash functions by up to four orders of magnitude. This substantial acceleration in hash computation offers immense potential for use cases requiring rapid verification of large virtual disk images, particularly in virtualization and software-defined storage.

1 INTRODUCTION

In the realm of virtualization, efficient disk space management is paramount for resource utilization. One approach is the utilization of sparse disk images for virtual disks. Sparse disk images offer a flexible and efficient means of disk allocation, particularly beneficial in scenarios where disk space conservation and dynamic allocation are priorities. Sparse disk images differ from pre-allocated disk images in their allocation strategy. Rather than pre-allocating the entire disk space upon creation, sparse disk images dynamically allocate storage space as data is written, utilizing only the space necessary to store actual data. Unallocated areas in the file are represented by file metadata to minimize storage space. This dynamic allocation makes sparse disk images particularly advantageous in environments where disk space is at a premium.

Virtual disk images are typically sparse. A virtual machine that is reading from a sparse virtual disk is oblivious to the fact that the disk is sparse and unallocated areas are seen as areas full of zeros (null bytes). A sparse virtual disk can be stored as a sparse file on a file system supporting sparseness, or a non-sparse file using image format supporting sparseness.

Virtual disk images are mostly empty. When provisioning a new virtual machine we install an operating system into a completely empty disk. While the virtual machine is running more data is added, however discarding deleted data can punch holes in the image. If the disk becomes too full it can be extended, adding large unallocated areas. Typically large portions of the image remain unallocated for the entire lifetime of the virtual machine. Figure 1 shows a typical disk image space allocation.

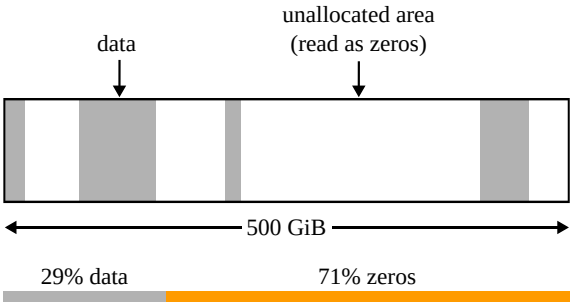

Figure 1: A typical sparse virtual disk image. In this example 71% of the image is unallocated.

[a] https://orcid.org/0009-0001-9265-7792
[b] https://orcid.org/0000-0001-5634-5436

Soffer, N. and Waisbard, E.
An Efficient Hash Function Construction for Sparse Data.
DOI: 10.5220/0012764500003767

In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 698-703
ISBN: 978-989-758-709-2; ISSN: 2184-7711

Disk utility tools are aware of image sparseness and take advantage of it when processing disk images. When a tool such as *qemu-img* (Bellard and the QEMU Project developers, 2003) copies a disk image, it first detects the allocated areas in the image using file system or image metadata. Using this info it skips reading unallocated areas from storage. Furthermore, when reading allocated areas, it uses zero detection to discover areas full of actual zeros, and treat them as unallocated areas. When writing to the target image, it can use efficient system calls to write zeros to storage. The entire software stack works in concert to enable efficient handling of zeros, leading to dramatic speed up and minimize I/O load when reading or writing sparse images.

However, existing checksum[1] tools like *sha256sum*, using the *SHA256* algorithm (Hansen and 3rd, 2006), are not aware of sparseness, and do not take advantage unallocated areas in the image or areas full of zeroes. Such tools read the entire image from storage, possibly transferring gigabytes of zeros over the wire when using remote storage. Then they compute a hash for the entire image, bit by bit. They do the same work regardless if the image is completely empty or completely full, which makes them very slow for typical virtual disk images.

Virtual disk images are commonly published as a compressed non-sparse image. A checksum is created using cryptographic hash function and published as well for verifying a downloaded image. However when one transfers the downloaded image to an actual storage in a virtualization system, the disk content is not stored in the same format, and the checksum of the downloaded image cannot be used to verify the image in the virtualization system. To verify an image using a checksum, you must compute a checksum of the image content, the same data as seen by the virtual machine using the image, and not a checksum of the box holding the image data. To do this efficiently, we need a hash function supporting sparse data. This is illustrated in Figure 2.

In the past computing a hash was considered much faster than copying a file, but due to improvements in network and storage, copying a file can be around 3 times faster than computing a hash[2]. Bearing in mind that the hash has to be computed over the entire disk image, while the copy operation is done only

[1]The term checksum is often used to describe the operation of computing a succinct representation value and it is commonly computed using a cryptographic hash function. In this paper when we use the term checksum, we refer to a computation of a cryptographic hash function.

[2]Recent NVMe devices provides read/write throughput of 6 GiBs, while the best hardware accelerated SHA256 can achieve at most 2 GiB/s

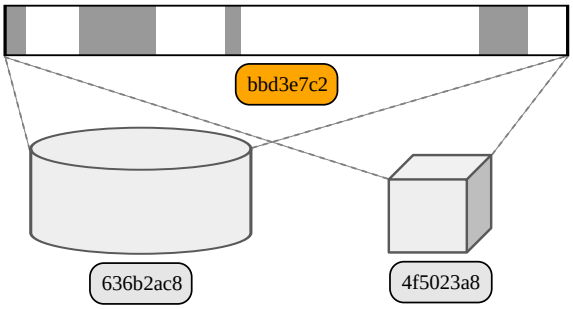

Figure 2: Two identical disk images with different physical representation. Computing a checksum over the physical representation of the image yields different values, while computing a checksum over the logical representation yields the same.

over the allocated parts significantly increases that gap. Namely, if the image is 80% empty then the data on which the hash is computed is 5 times larger and computing the hash can be 15 times slower.

These days most computing devices, including entry level phones, have multiple cores. Large servers can have up to hundreds of cores. However the state of the art cryptographic hash functions like *SHA256* and *SHA3-256* (Dworkin, 2015) use only a single core because the algorithm is is inherently sequential and cannot be parallelized to leverage the multiple cores. Consequently, these hash functions, that have to go over the entire image are limited in their computing power to a single core. Recent algorithms like *BLAKE3* (O'Connor et al., 2019) can use all available cores when using regular file via memory mapping, but use only one core in other cases, for example when reading from a block device or a pipe.

We propose a new hash construction optimized for sparse virtual disk images that is up to *4 orders of magnitude more efficient*, which results in being both faster and energy efficient compared with state of the art cryptographic hash functions. Our most important contribution is an efficient way to update the hash with zeros - unallocated areas in the image, without reading anything from storage, or adding any data to the hash. When adding actual data to the hash, we use fast zero detection to treat blocks full of zeros as unallocated area, eliminating the computation. In addition, our construction allows parallel processing, that scales linearly with the number of threads.

Our solution is a modular construction that turns any secure hash function into a hash function that works efficiently with sparse input. This modular construction that uses two layers enables using either the same hash function on both the *inner* and *outer* layers or using different ones. Using different hash functions allows enhancing security or tuning performance by adding a stronger or faster hash function.

2 THE CONSTRUCTION

The *blkhash* (Soffer, 2021) construction is designed to work efficiently with sparse disk images. Unlike common hash functions that go over the entire image sequentially, including the unallocated areas, *blkhash* works more efficiently by:

1. Minimizing the computation over unallocated blocks or blocks full of zeros.
2. Computing hashes of data blocks in parallel.

Loosely speaking, the *blkhash* construction is utilizing a two levels *Merkle tree* (Merkle, 1988) construction. On the first level, we split the input image into fixed sized blocks and compute the hash value of every block. In the second level, we perform another hash computation of all the hashed values of the blocks in the order in which they were split and the result is the output value of *blkhash*. This is illustrated in Figure 3

This enables performing the computation in parallel and utilizing all the available cores. We note that if two blocks are of the same value then their hash value is the same. As a result, we do not need to compute the hash value of the all-zero block repeatedly. In fact, we can pre-compute the hash value of an all-zero block in advance.

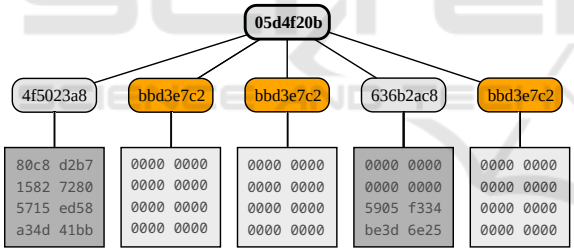

Figure 3: The blkhash construction with a an example image with 5 blocks. We can see that 3 blocks are full of zeros and have the same hash value. The *blkhash* algorithm eliminates the computation of the zero blocks.

We now describe the construction more formally. Let us denote by H our *blkhash* function that uses two collision resistant hash functions h_{inner} and h_{outer}. In practice, the *inner* and *outer* hash functions are expected to be the same function, but they can also differ and we discuss this case later. Let $x \in \{0,1\}^*$ be the input to H. We denote by l the length of x in bytes. We set the block size to k and we split x into blocks of size k. We note that if the length of x is not a multiplicity of k, then the final block will be shorter than k. We calculate the number of blocks $n = \lceil \frac{l}{k} \rceil$.

The resulting split looks as follows:

$$x = \underbrace{x_1}_{k} || \ldots || \underbrace{(x_{n-1})}_{k} || \underbrace{x_n}_{\leq k}$$

where $n-1$ blocks are of size k and the final block may be shorter than k.

We compute *blkhash* H as follows:

$$H(x) = h_{outer}(h_{inner}(x_1)|| \ldots ||h_{inner}(x_n)||l)$$

Namely, we hash each of the blocks separately using H_{inner} and then hash the resulting values in the original order along with the length of x using H_{outer}. This construction enables parallel computation for the *inner* block hashes, since computing a hash of one block does not depend on the hash of the previous blocks. This enables linear scaling with number of threads computing the block hashes.

Blocks that are unallocated or full of zeros results in the same hash value, and can use a pre-computed zero block hash value.

Input :
H_{outer} : collision resistant hash
H_{inner} : collision resistant hash
k : block size
x : message to hash
Output: Hash value of message x
$h_{zero} \leftarrow H_{inner}$(zero block of length k);
$i \leftarrow 0$;
while $i < |x|$ **do**
  $x_i \leftarrow x[i, i+k]$;
  if $|x_i| = k$ *and* x_i *is a zero block* **then**
    add h_{zero} to H_{outer};
  else
    $h_i \leftarrow H_{inner}(x_i)$;
    add h_i to H_{outer};
  end
  $i \leftarrow i+k$;
end
add $|x|$ to H_{outer};
return the result of H_{outer} evaluation;

Algorithm 1: The blkhash construction.

Detecting zero blocks is done in 2 ways:

1. Detect the unallocated areas in the image from file system or image metadata, avoiding reading the data from storage and eliminating all the computation. This is the most important optimization, speeding up processing by multiples orders of magnitude.
2. Efficiently detect blocks full of zeros (e.g. using *memcmp*) and avoiding the computation of block hashes. Scanning blocks for zeros is faster than computing a hash, even when using a fast cryptographic hash such as *BLAKE3*, that can take advantage of widest SIMD instructions.

Note that zero block optimization only affects the performance of computing the *inner* hash. The com-

putation yields the same hash value regardless of the efficiency of the computation.

3 PROOF OF SECURITY

Collision resistance is a fundamental property of cryptographic hash functions. Collision resistance guarantees that it is computationally infeasible to find two distinct inputs that hash to the same output value. This property is vital for maintaining data integrity as it prevents malicious actors from producing two different files with identical hash values [3].

In this section we prove that if the underlying hash functions h_{inner} and h_{outer} are collision resistant then so is our construction.

Assume toward contradiction that one can find two inputs x and x', such that $H(x) = H(x')$, then we show a collision either for h_{inner} or h_{outer}.

We split our proof into two cases:

- **Case 1:** x and x' are of different length
- **Case 2:** x and x' are of the same length

In **case 1** since the length is part of the input to the h_{outer}, then the input for h_{outer} is different when x and x' are of different length. Thus, if $H(x) = H(x')$ then we have a collision in the *outer* hash function.

More formally, let us denote by l the length of x and by l' the length of x'.

$$\begin{aligned} H(x) &= h_{outer}(\cdots||l) \\ &= h_{outer}(\cdots||l') = H(x') \end{aligned} \tag{1}$$

and since $l \neq l'$, if $H(x) = H(x')$ then we get a collision for h_{outer}.

In **case 2**, we focus on block i in which $x_i \neq x'_i$, noting that there has to be at least one such block, otherwise x and x' are identical.

If $h_{inner}(x_i) = h_{inner}(x'_i)$ then we have a collision for h_{inner}.

If $h_{inner}(x_i) \neq h_{inner}(x'_i)$ then we get that

$$\begin{aligned} H(x) &= h_{outer}(\cdots||h_{inner}(x_i)||\cdots||l) \\ &= h_{outer}(\cdots||h_{inner}(x'_i)||\cdots||l) = H(x') \end{aligned} \tag{2}$$

and we got a collision for h_{outer}. □

[3]Consider an attacker that can create two files, one benign and one containing malware, that result in the same hash value, then he can get the benign version signed by a trusted authority and then have the malware version distributed along with the same signature.

4 SPECIFICATION

Here we specify how a single threaded *blkhash* hash function can be implemented.

The construction requires the following parameters. Changing any of the parameters changes the construction and the hash value.

- *outer-hash-algorithm* - a collision resistant hash algorithm.
- *inner-hash-algorithm* - a collision resistant hash algorithm.
- *block-size* - block size in bytes. A power of 2, equal or larger than 64 KiB is recommended to match common image formats internal structure.

The construction must maintain the following state:

- *outer-hash* - an instance of *outer-hash-algorithm*. The hash must be initialized before feeding data into the hash function.
- *input-length* - if the input length in bytes is unknown when creating the hash, initialize it to 0, and update it incrementally when feeding data into the hash.

To implement zero optimization (as noted before, zero optimization is optional), the construction must also maintain the following state:

- *zero-block-hash* - a hash value of an all zero block of length *block-size* bytes, computed using *inner-hash-algorithm*.

Split the input of the hash function to fixed size blocks of *block-size* bytes. If the input length is not a multiple of the block size, the last block may be shorter than *block-size*, but it cannot be empty. If the input length is zero no block need to be processed.

For each input block perform the following operations:

1. If the block length is equal to *block-size* and zero optimizations are implemented, check if the block contents are zeros. We have 2 cases:
   - If file system or image metadata are available, and the image is known to read as zeros.
   - Otherwise if no metadata is available, check if the block is full of zeros.

   If block contents are zeros, update *outer-hash* with the pre-computed *zero-block-hash* value.
2. Otherwise compute a hash value of the block using the *inner-hash-algorithm*, and update the *outer-hash* with the computed hash value.

When all input blocks were processed, update the *outer-hash* with *input-length* as a 64 bit little-endian integer.

Finalize the *outer-hash*, producing the hash value. This is the *blkhash* hash value of the input.

5 EMPIRICAL RESULTS

We measured the throughput of *blkhash* hash function using both *BLAKE3* and *SHA256* provided by *openssl* (The OpenSSL Project, 2003) for the outer and inner hash functions. *SHA256* is considered the industry standard and recent CPUs also feature hardware acceleration of it. *BLAKE3* is an extremely fast hash function on 64-bit platform supporting *AVX-512* instructions. These functions demonstrate how *blkhash* adapts the most widely used cryptographic hash functions into sparse optimized hash functions.

We use the notation *blk-ALGORITHM* to describe application of *blkhash* using *ALGORITHM* for the outer and inner hash functions.

Real disk images are typically comprised of three types of data and *blkhash*'s performance varies according to it. We generated these input types and measured how *blkhash* performs on each of them. The three types are:

- *data*: all blocks in the input are non-zero. This is the worst case where *blkhash* must compute a hash for all blocks.
- *zero*: all blocks in the input contain only zeros. This is a better case, where all blocks must be scanned to detect zeros, but no hash is computed for any block.
- *hole*: all blocks are unallocated. This is the best case where no data is scanned and no hash is computed for any block.

We ran the tests on two *AWS* bare metal instances:

- *c7i.metal-24xl* (Amazon Web Services, 2023b) powered by 4th Generation Intel Xeon Scalable processor (Sapphire Rapids 8488C), featuring 48 cores and 96 vCPUs. We tested with Hyper-Threading disabled since it is not a good match for this type of workload.
- *c7g.metal* (Amazon Web Services, 2023a) powered by Arm-based AWS Graviton3 processors, featuring 64 cores.

We measure using the *blkhash-bench* (Soffer, 2022) program, providing an easy to use command line interface to measure any input type with any configuration supported by the *blkhash* library. The program allocates a fixed size pool of buffers and feed the data as fast as possible to the *blkhash* hash function without doing any I/O. Actual results with real images will be much lower since reading data from storage is typically the bottleneck.

To reproduce our results please refer to the benchmarking documentation in the *blkhash* repository: https://gitlab.com/nirs/blkhash/-/blob/paper/docs/benchmarking.md

5.1 Zero Optimization

This benchmark shows the effect of zero optimization on the hash throughput when using different algorithms for the internal hash functions. We focus on the fastest algorithms for the tested machine, *BLAKE3* on Intel Xeon and *SHA256* on AWS Gravitron3, using SIMD instructions or crypto extensions.

Figure 4 shows *blkhash* throughput on *AWS c7i.metal-24xl* instance using *BLAKE3* for the outer and inner hash functions. Hashing unallocated areas (hole) reached the maximum throughput with 1 thread, 2223 times faster than single threaded *BLAKE3*. Hashing blocks full of zeros (zero) is up to 64.3 times faster than single threaded *BLAKE3*. Hashing blocks full of non-zero bytes (data) is up to 33.6 times faster than single threaded *BLAKE3*.

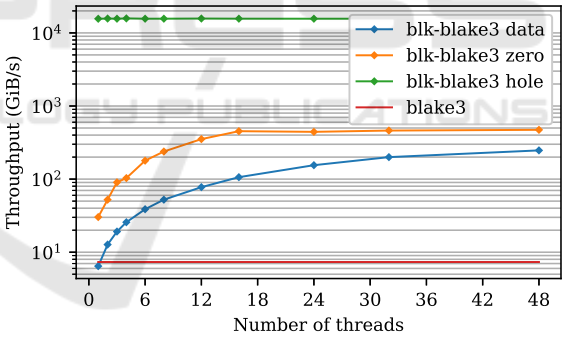

Figure 4: Throughput of blk-blake3, higher is faster.

Figure 5 shows *blkhash* throughput on *AWS c7g.metal* instance using *SHA256* for the outer and inner hash functions. Hashing unallocated areas (hole) reached the maximum throughput with 1 thread, 15323 times faster than single threaded *SHA256*. Hashing blocks full of zeros (zero) is up to 270.9 times faster than single threaded *SHA256*. Hashing blocks full of non-zero bytes (data) is up to 62.9 times faster than single threaded *SHA256*.

5.2 Carbon Footprint

We measured the throughput in cycles per byte as a good proxy for amount of energy used to compute a

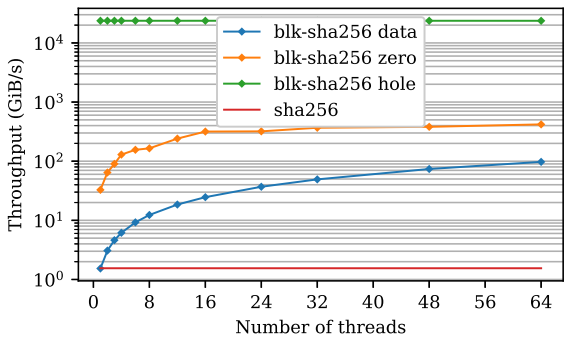

Figure 5: Throughput of blk-sha256, higher is faster.

hash. Zero optimization not only speeds up hash computation by up to 4 orders of magnitude but it also lowers the carbon footprint of the computation by 4 orders of magnitude - at the same time.

Figure 6 shows *blkhash* efficiency on *AWS c7i.metal-24xl* instance using *BLAKE3* for the outer and inner hash functions. Hashing unallocated areas (hole) shows throughput of 0.0002 cycles per byte, 2400.0 times lower than *BLAKE3*. Hashing blocks full of zeros (zero) shows constant throughput of 0.1 cycles per byte for any number of threads, 4.8 times lower than single threaded *BLAKE3*. Hashing blocks full of non-zero bytes (data) show constant throughput of 0.54 cycles per bytes for any number of threads, 1.12 times higher than single threaded *BLAKE3*.

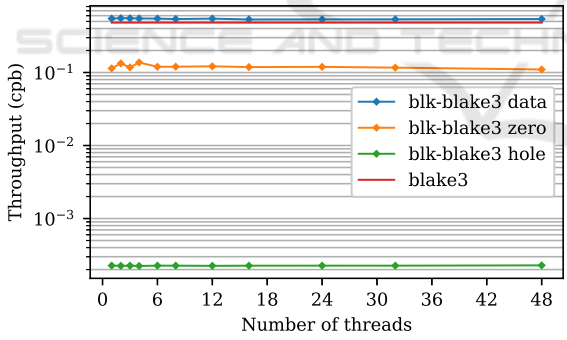

Figure 6: Throughput of blk-blake3 in cycles per byte, lower is better.

6 CONCLUSIONS

As the world increasingly shifts into the cloud, the demand for secure hash functions of high throughput becomes more pronounced than ever before. The traditional approach to verifying file integrity through hash computation has long been plagued by inefficiencies when dealing with large files. The perception that computing the hash value is negligible compared to the time it takes to copy a file, no longer holds in modern computing. Optimizing the performance over sparse disk images needs to consider hash computation in addition to the copying operation.

The introduction of *blkhash* marks a new direction in the realm of hash function design. We address these challenges head-on by minimizing the computational overhead associated with empty or unallocated areas within the file and also by leveraging the multi core technology by parallelization of the computation. An important feature of blkhash is its modular design, which allows it to utilize any existing hash function as a building block. Whether it be well-established standards like *SHA256* or modern alternatives like *BLAKE3*, *blkhash* seamlessly integrates these hash functions into its framework. This modular approach not only enhances the flexibility and versatility of *blkhash*, but also leverages the proven security properties of established hash algorithms. We provide a reference implementation along with a suite of benchmarks. Our results reveal that *blkhash* achieves acceleration levels of up to four orders of magnitude, positioning it as a game-changer for use cases requiring rapid verification of large virtual disk images.

REFERENCES

Amazon Web Services (2023a). Amazon ec2 c7g instances. https://aws.amazon.com/ec2/instance-types/c7g/.

Amazon Web Services (2023b). Amazon ec2 c7i instances. https://aws.amazon.com/ec2/instance-types/c7i/.

Bellard, F. and the QEMU Project developers (2003). Qemu disk image utility. https://www.qemu.org/docs/master/tools/qemu-img.html.

Dworkin, M. (2015). Sha-3 standard: Permutation-based hash and extendable-output functions.

Hansen, T. and 3rd, D. E. E. (2006). US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634.

Merkle, R. C. (1988). A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology–CRYPTO '87: Proceedings*, page 369–378. Springer-Verlag.

O'Connor, J., Aumasson, J.-P., Neves, S., and Wilcox-O'Hearn, Z. (2019). Blake3 - one function, fast everywhere. https://github.com/BLAKE3-team/BLAKE3-specs.

Soffer, N. (2021). blkhash - block based hash optimized for disk images. https://gitlab.com/nirs/blkhash/-/tree/paper.

Soffer, N. (2022). blkhash-bench - blkhash benchmark tool. https://gitlab.com/nirs/blkhash/-/blob/paper/test/blkhash-bench.c.

The OpenSSL Project (2003). OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org.