

Intel[®] QuickAssist Technology (Intel[®] QAT) and OpenSSL-1.1.0: Performance

White Paper

January 2018

Revision 001



You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to www.intel.com/benchmarks.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or visit www.intel.com/design/literature.htm. No computer system can be absolutely secure.

Intel, Xeon, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2018, Intel Corporation. All rights reserved.



Contents

1	Introduction.....	6
	1.1 Terminology.....	6
2	Asynchronous Operations.....	7
	2.1 Motivation: Design for Performance.....	7
	2.2 Why Async?.....	8
	2.3 Design.....	8
	2.4 ASYNC_JOB Infrastructure.....	9
	2.5 ASYNC Event Notification.....	10
3	Additional Performance Optimizations.....	11
	3.1 Pipelining.....	11
	3.2 Pseudo Random Function (PRF).....	11
	3.3 Intel® QuickAssist Technology (Intel® QAT) Engine.....	12
	3.4 Big/Small Request Offload.....	13
4	Performance.....	14
	4.1 Benchmark and Results.....	14
	4.2 Algorithmic Performance (openssl speed).....	14
	4.3 Symmetric Algorithm Performance.....	15
	4.4 Application-Level Benchmark (NGINX-1.10 + OpenSSL-1.1.0).....	16
	4.5 Benchmark Topology.....	16
5	Conclusion.....	21
Appendix A	Platform Details.....	22



Figures

Figure 1. Synchronous Execution	7
Figure 2. Asynchronous Execution.....	8
Figure 3. Intel® QuickAssist Technology Stack Diagram	9
Figure 4. OpenSSL-1.1.0 ASYNC_JOB Processing Flow.....	10
Figure 5. Client-Server Benchmark Overview	16
Figure 6. Web-Server Physical Topology	17
Figure 7. RSA-2K Connections per Second (NGINX-1.10 + OpenSSL-1.1.0)	18
Figure 8. ECDHE-RSA-2K Connections per Second (NGINX-1.10 + OpenSSL-1.1.0)	18
Figure 9. ECDHE-ECDSA Connections per Second (NGINX-1.10 + OpenSSL-1.1.0)	19
Figure 10. Comparison of Synchronous versus Asynchronous Infrastructure.....	20

Tables

Table 1. Terminology	6
Table 2. Performance of RSA 2K with openssl speed.....	15
Table 3. Performance of ECDSA-P256 with openssl speed	15
Table 4. Performance of ECDH-P256 Compute Key with openssl speed	15
Table 5. Commands used for Chained Cipher (aes-128-cbc-hmac-sha1)	15
Table 6. Chained Cipher: aes-128-cbc-hmac-sha1 (Gbps).....	15
Table 7. TLS Benchmark Test Parameters	17
Table 8. References	23



Revision History

Document Number	Revision Number	Description	Revision Date
337003	001	Initial release.	January 2018

§



1 Introduction

Transport Layer Security (TLS) is the backbone protocol for Internet security today. It provides the foundation for expanding security everywhere within the network. Security is an element of networking infrastructure that must not be underemphasized or taken for granted. Security is critical to the foundation of networking. Adding security into existing infrastructures generally comes with a trade-off between cost and performance.

With the addition of a new class of features added into OpenSSL-1.1.0, Intel has been able to significantly increase performance for asynchronous processing with Intel® QuickAssist Technology (Intel® QAT). This paper explores the design and usage of these features:

- ASYNC_JOB infrastructure
- ASYNC event notifications
- Pipelining
- PRF engine support

This paper will demonstrate how the combination of these features with Intel® QAT results in tangible performance gains, as well as how an application can utilize these features at the TLS and EVP level.

1.1 Terminology

Table 1. Terminology

Term	Description
API	Application Programming Interface
BSD	Berkeley Software Distribution
ECDH	Elliptic Curve Diffie Hellman
ECDHE	Elliptic Curve Diffie Hellman Ephemeral
ECDSA	Elliptic Curve Digital Signature Algorithm
EVP	EnVeloPe
Intel® QAT	Intel® QuickAssist Technology
IV	Initialization Vector
RFC	Request For Comments
RSA	Rivest–Shamir–Adleman
PRF	Pseudo Random Function
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security

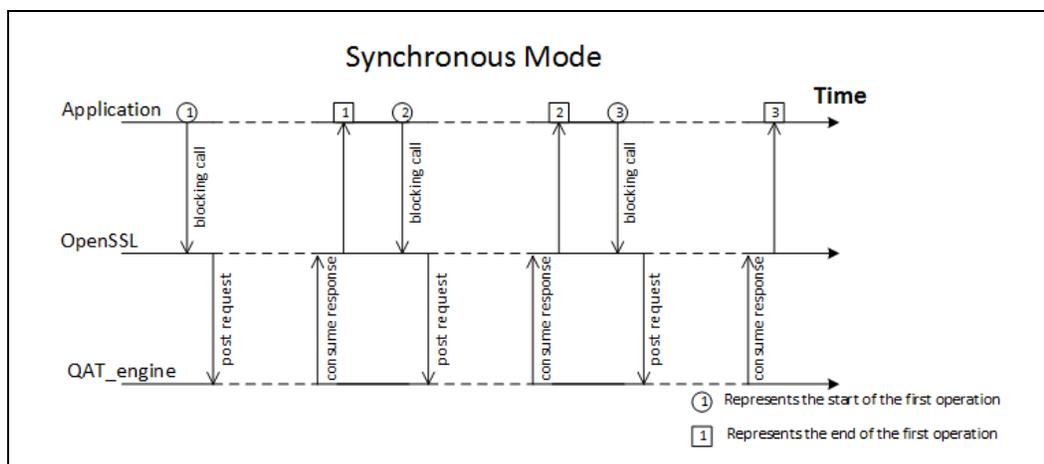
2 Asynchronous Operations

2.1 Motivation: Design for Performance

The asynchronous infrastructure added into OpenSSL-1.1.0 provides the capability for cryptographic operations to execute asynchronously with respect to the stack and application. Generically the infrastructure could be applied to any asynchronous operations that might occur, but currently only encompasses cryptographic operations executed within the engine framework.

For the context of this paper, we will define asynchronous operations as those that occur independently of the main program's execution. These operations will be initiated and consumed (using events/polling) by the main program, but will occur in parallel to those operations. The following two figures are illustrations of the shift in execution.

Figure 1. Synchronous Execution

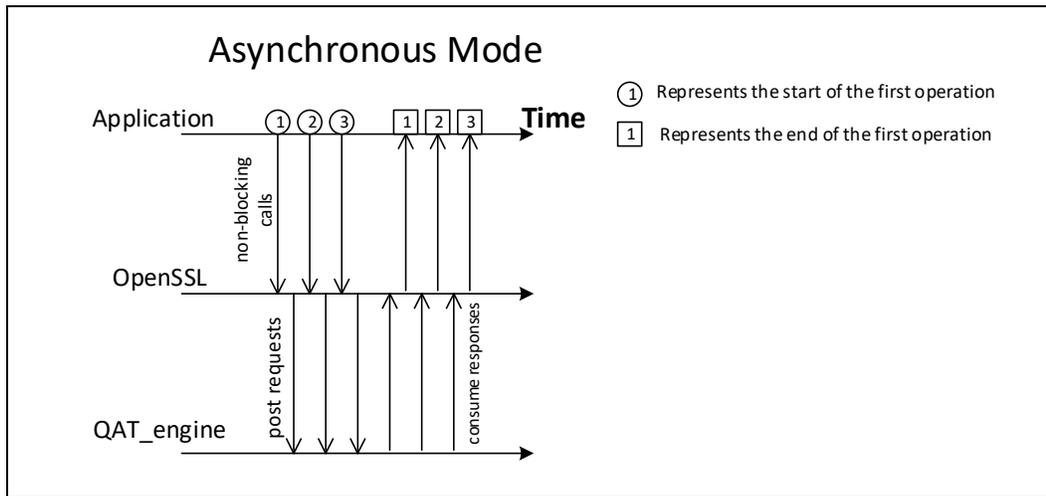


Synchronous mode of operation forces a single API call to be blocking until the completion of the request. When a parallel processing entity is part of the flow of execution, there will be times when the processor is not processing data. These are represented by dashed lines in [Figure 1](#) and effectively result in missed opportunities for increased performance. From the application perspective, this results in blocking at the API. When using a separate accelerator underneath this API, the application can perform a busy-loop while waiting for a response from the accelerator, or context switch using execution models similar to pthreads to allow other useful work to be accomplished while waiting. However, both solutions are costly. Polling consumes processor cycles and prevents multiple operations from running in parallel. While threading allows parallelism and more effectively uses processor cycles, most high level context management libraries like pthreads come with a heavy cycle cost to execute and manage.

The asynchronous programming model increases performance by making use of these gaps. It also enables parallel submission more efficiently using a parallel processing entity (for example, Intel® QuickAssist Technology).

Intel® QAT provides acceleration of cryptographic and compression calculations on a separate processing entity, processing the requests asynchronously with respect to the main program. Having an asynchronous processing model in OpenSSL-1.1.0 allows for more efficient use of those capabilities, as well as increased overall performance.

Figure 2. Asynchronous Execution



2.2 Why Async?

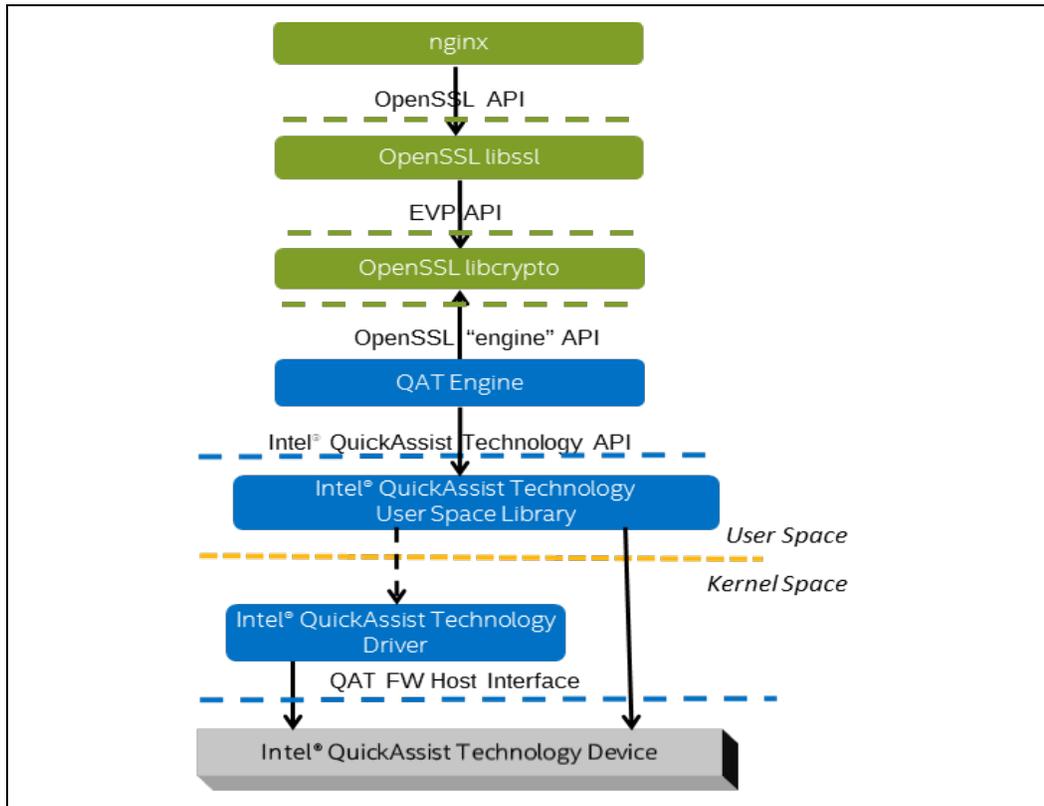
To efficiently use acceleration capabilities, a mechanism to allow the application to continue execution while waiting for the Intel® QAT accelerator to complete outstanding operations is required. This programming model is very similar to nonblocking Berkeley Software Distribution (BSD) sockets; operations are executed outside the context of the main application, allowing the application to make the best use of available processor cycles while the accelerator is processing operations in parallel. This capability is controlled by the application, which must be updated to support the asynchronous behavior, as it has the best knowledge of when to schedule each TLS connection. [Figure 10](#) demonstrates increased performance as a result of centralizing the scheduling entity in the application.

2.3 Design

The Intel® QuickAssist Technology accelerator is accessed through a device driver in kernel space and a library in user space. Cryptographic services are provided to OpenSSL through the standard engine framework. This engine (refer to [Table 8, \[11\]](#)) builds on top of the user space library, interfacing with the Intel® QAT API, which allows it to be used across Intel® QAT generations without modification. This layering and integration into the OpenSSL framework allows for seamless utilization by applications. The addition of asynchronous support into OpenSSL-1.1.0 means that the application can also drive higher levels of performance using a standardized API.



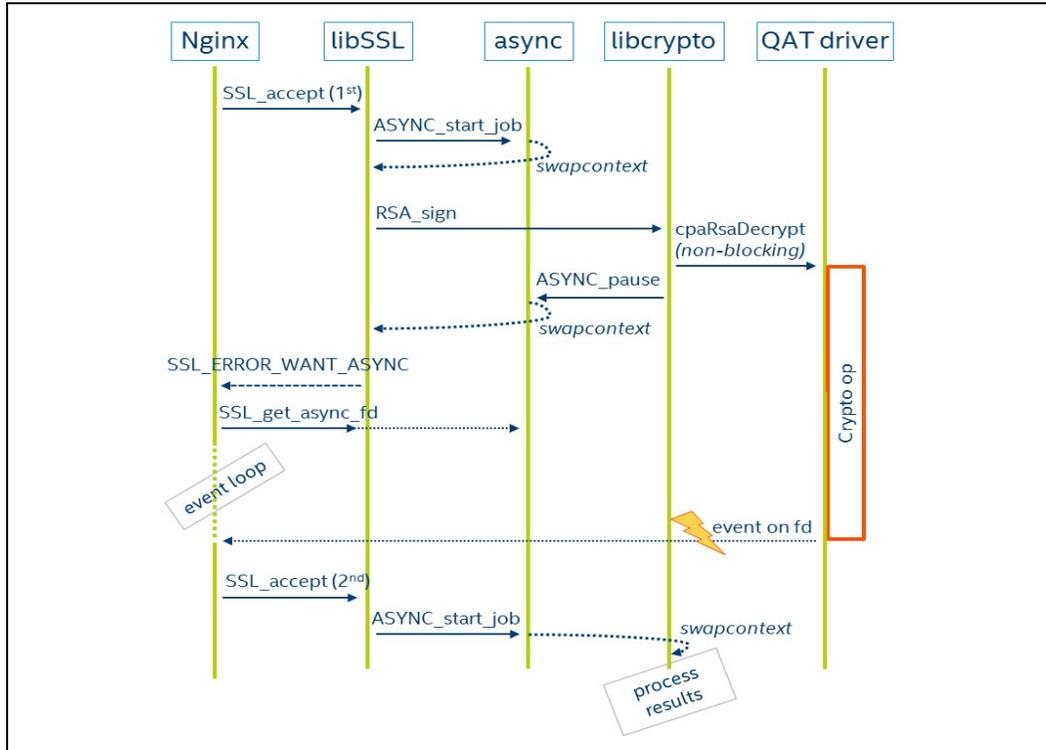
Figure 3. Intel® QuickAssist Technology Stack Diagram



2.4 ASYNC_JOB Infrastructure

The ASYNC_JOB infrastructure is built on a number of primitives to allow the creation and management of lightweight execution contexts. The infrastructure added to OpenSSL-1.1.0 (refer to [Table 8, \[2\]](#)) provides all the necessary functions to create and manage ASYNC_JOBS (similar in concept to fibers or co-routines) but does not actively manage these resources. Management is left to the user code leveraging this capability. Logically, the ASYNC_JOB infrastructure is implemented as part of the crypto complex in OpenSSL-1.1.0, namely libcrypto, and is utilized by the TLS stack. This allows applications to continue to use the well-known OpenSSL APIs in the same manner as before, utilizing ASYNC_JOBS where possible in the application. The ASYNC_JOBS are publicly accessible APIs in OpenSSL-1.1.0 and as such, the application can also use them directly in conjunction with the EVP APIs or indirectly through the OpenSSL-1.1.0 APIs.

Figure 4. OpenSSL-1.1.0 ASYNC_JOB Processing Flow



The function call flow in [Figure 4](#) shows one usage scenario from the top level `SSL_accept (1st)` call. When an application identifies a TLS connection as being asynchronous capable, standard OpenSSL calls will grab an `ASYNC_JOB` context, thereby allowing the underlying layers of the stack to pause execution, in this example in the Intel® QAT engine. This results in the function returning to the application with the error status `SSL_ERROR_WANT_ASYNC`. The application can then register for a file descriptor (FD) associated with this TLS connection, and use the standard `epoll/select/poll` calls to wait for availability of a response. Once the application is notified, it can call the associated OpenSSL API, `SSL_accept (2nd)` again with that TLS connection, thereby completing the response processing. Alternatively, the application can forego using the FD and event notifications, instead of continuously invoking the top level OpenSSL API until a successful response is returned.

2.5 ASYNC Event Notification

OpenSSL-1.1.0 includes a notification infrastructure to signal when to resume the execution of asynchronous crypto operations. The notifications from the crypto engine to the application are delivered using events on file descriptors that can be managed using the APIs provided by OpenSSL. This provides an abstraction layer that is independent of both the application and the particular hardware accelerator being used. The file descriptor is owned by the component that originates the event (in this case, the engine implementation). This allows the originator to define how they want to create and possibly multiplex signals in case there are multiple sources.



3 Additional Performance Optimizations

3.1 Pipelining

Pipelining allows an engine to perform multiple symmetric crypto operations in parallel on a single TLS connection, increasing the throughput of bulk transfers. When pipelining is enabled for a TLS context, the input buffer of each `SSL_write` operation is split into multiple independent records that can be processed simultaneously by the engine. The results of the operation are then written to the socket in the correct order, which is transparent to the client. The alternate direction (`SSL_read`) is also supported where sufficient data is available.

Pipelining provides the greatest benefits when the number of concurrent connections to the server/endpoint are small. In this scenario, the parallelizing crypto operations for each individual connection leads to a better utilization of the crypto engine. When the number of simultaneous TLS connections is large, parallelization comes from the application's ability to maintain a large number of connections. In summary, there are two dimensions to achieve parallelization: at the individual connection level, and loading across connections.

Versions of OpenSSL prior to 1.1.0 had a similar functionality called multi-block, but pipelining provides two significant improvements over the previous implementation:

1. Pipelining is not limited to four or eight buffers, but it can be used with an arbitrary number of buffers (for example, pipes).
2. The engine is no longer responsible for creating the headers of the record, hence pipelining is not dependent on a particular protocol (for example, TLS).

To parallelize the encryption of TLS records, they must be independent from a cryptographic perspective. For this reason, pipelining is only supported for TLSv1.1 and TLSv1.2, where the IV is explicit, and does not depend on the previous record. There is currently no support for SSLv3, TLSv1.0, or DTLS (all versions) in the OpenSSL-1.1.0 branch.

3.2 Pseudo Random Function (PRF)

Pseudo Random Function (PRF) is used during the TLS handshake for the expansion of secrets to subsequently be used for the purposes of key generation or validation. The function definition varies across the TLS versions and is defined in the relevant RFCs (refer to [Table 8, \[3\]](#)).

In OpenSSL 1.1.0, the PRF derive operation is exposed with a new API (refer to [Table 8, \[4\]](#)) at the EVP level, and can be offloaded to the engine as a single request.



This is an important change from previous versions where the operation was actually performed as a sequence of digest operations. Although the software implementation has not changed, the new API allows the application to decrease the number of requests to a hardware accelerator down to one, with a significant reduction in overhead for offload.

A new algorithm has been added to implement the key derivation function for the TLS protocol (EVP_PKEY_TLS1_PRF). The following functions can be used to set the message digest (for example, EVP_sha256()), the secret and the seed used for the key derivation:

```
int EVP_PKEY_CTX_set_tls1_prf_md(EVP_PKEY_CTX *pctx, const EVP_MD *md);
int EVP_PKEY_CTX_set1_tls1_prf_secret(EVP_PKEY_CTX *pctx, unsigned char
*sec, int seclen);
int EVP_PKEY_CTX_add1_tls1_prf_seed(EVP_PKEY_CTX *pctx, unsigned char
*seed, int seedlen);
```

They are integrated into libSSL (refer to [Table 8, \[5\]](#)) and are automatically used for the TLS handshake. For more information, refer to the official documentation (refer to [Table 8, \[6\]](#)).

3.3 Intel® QuickAssist Technology (Intel® QAT) Engine

As shown in [Figure 3](#), the Intel® QAT engine for OpenSSL 1.1.0 improves the performance of secure applications by offloading the computation of cryptographic operations while freeing up the processor to perform other tasks. The engine supports the traditional synchronous mode for compatibility with existing applications, as well as the new asynchronous mode introduced in OpenSSL 1.1.0 to achieve the best possible performance.

Once the engine has been loaded and initialized, all crypto operations that have been registered and executed via the EVP API will be offloaded transparently to Intel® QAT engine. This gives access to the performance improvement of Intel® QAT, while significantly reducing the time and cost required to integrate the APIs into a new application. The code is freely available on GitHub (refer to [Table 8, \[11\]](#)).

By default, the engine offloads the following crypto algorithms to hardware accelerators:

- Asymmetric PKE Offload
 - RSA Support with PKCS1 Padding for Key Sizes 1024/2048/4096.
 - DH Support for Key Sizes: 768/1024/1536/2048/3072/4096.
 - DSA Support for Key Sizes: 160/1024, 224/2048, 256/2048, 256/3072.
 - ECDH Support for the following curves:
 - NIST Prime Curves: P-192/P-224/P-256/P-384/P-521.
 - NIST Binary Curves: B-163/B-233/B-283/B-409/B-571.
 - NIST Koblitz Curves: K-163/K-233/K-283/K409/K-571.
 - ECDSA Support for the following curves:
 - NIST Prime Curves: P-192/P-224/P-256/P-384/P-521.
 - NIST Binary Curves: B-163/B-233/B-283/B-409/B-571.
 - NIST Koblitz Curves: K-163/K-233/K-283/K409/K-571.
- Symmetric Chained Cipher Offload:
 - AES128-CBC-HMAC-SHA1/AES256-CBC-HMACSHA1.
 - AES128-CBC-HMAC-SHA256/AES256-CBC-HMACSHA256.
- Pseudo Random Function (PRF) offload.



3.4 Big/Small Request Offload

Choices may be developed for optimizing performance of small payloads that may incur a larger offload cost versus cost of software implementation.

Given a particular crypto operation, the user can set big/small request thresholds. If a value falls below that threshold, the request would not be offloaded. The appropriate considerations will depend on the speed of the processor, cost of offload, and whether you are optimizing for processor cycles or latency. The application may use the following custom engine control command to set the threshold:

`SET_CRYPTO_SMALL_PACKET_OFFLOAD_THRESHOLD.`

§



4 Performance

This chapter provides Intel benchmark test results.

4.1 Benchmark and Results

The work presented is focused on performance features added to OpenSSL-1.1.0 along with optimizations added throughout the stack to improve cryptographic throughput when utilizing Intel® QAT provided on the Intel® Xeon Scalable processor Platform Gold 6152 with the Intel® C627 Chipset. These performance improvements are concentrated on two different levels:

1. The algorithmic computation level (openssl speed) described in Algorithmic Performance (openssl speed).
2. The application TLS processing level (NGINX* built on OpenSSL-1.1.0) described in Application level benchmark (NGINX-1.10 + OpenSSL-1.1.0).

The performance gains from asynchronous features are measured by comparing benchmarks from the following three configurations:

- Software: Crypto operations calculated on the main processor using OpenSSL-1.1.0.
- Sync: Crypto operations offloaded to Intel® QAT and performed synchronously.
- Async: Crypto operations offloaded to Intel® QAT and performed asynchronously.

Note: Platform configuration details are provided in [Appendix A](#).

4.2 Algorithmic Performance (openssl speed)

OpenSSL comes with the command line utility `openssl speed` to measure algorithmic performance. The application interfaces directly to the EVP APIs of the crypto module in OpenSSL (library `libcrypto`) and executes the requested algorithm in a tight loop across a number of request sizes for a specified period of time. It then totals the number of completed operations and reports the resulting throughput in the context of the algorithm specified. For example, `aes-128-cbc-hmac-sha1` is reported in kilobytes per second, while RSA is reported as the number of verifies or signs per second. With the inclusion of `async`, the command line now takes the optional parameter `-async_jobs`. This specifies to the application how many jobs to create and execute in parallel. All the benchmarks are executed with the process explicitly affinity to a particular core to reduce the number of context switches between different cores. This is done manually using the command `taskset` to set the affinity.

The following commands were used to gather performance. All measurements for asymmetric algorithms were collected using a single core.



Table 2. Performance of RSA 2K with openssl speed

Mode	Command	Sign/s	Verify/s
Software	<code>./openssl speed -elapsed rsa2048</code>	957	34,483
Sync	<code>./openssl speed -elapsed -engine qat rsa2048</code>	2,257	15,625
Async	<code>./openssl speed -engine qat -elapsed -async_jobs 36 rsa2048</code>	100,000	200,000

Table 3. Performance of ECDSA-P256 with openssl speed

Mode	Command	Sign/s	Verify/s
Software	<code>./openssl speed -elapsed ecdsap256</code>	33,898	15,937
Sync	<code>./openssl speed -engine qat -elapsed ecdsap256</code>	1,689.2	932
Async	<code>./openssl speed -engine qat -elapsed -async_jobs 36 ecdsap256</code>	95,238	76,923

Table 4. Performance of ECDH-P256 Compute Key with openssl speed

Mode	Command	Operation/s
Software	<code>./openssl speed -elapsed ecdhp256</code>	24,096
Sync	<code>./openssl speed -elapsed -engine qat ecdhp256</code>	1,757.5
Async	<code>./openssl speed -engine qat -elapsed -async_jobs 36 ecdhp256</code>	142,857

4.3 Symmetric Algorithm Performance

The following commands have been used to measure the performance of the cipher suite (aes-128-cbc-hmac-sha1).

Table 5. Commands used for Chained Cipher (aes-128-cbc-hmac-sha1)

Mode	Command
Software	<code>./openssl speed -elapsed -evp aes-128-cbc-hmac-sha1 -multi 5</code>
Sync	<code>./openssl speed -elapsed -engine qat -evp aes-128-cbc-hmac-sha1 -multi 5</code>
Async	<code>./openssl speed -elapsed -async_jobs 75 -engine qat -evp aes-128-cbc-hmac-sha1 -multi 5</code>

Table 6. Chained Cipher: aes-128-cbc-hmac-sha1 (Gbps)

Mode	# cores	64B	256B	1kB	8kB	16kB
Software	5	11.87	15.5	18.42	19.93	20.12
Sync	5	0.04	0.16	0.63	5.07	10.12
Async	5	1.46	5.75	22.58	120.07	142.11

Note: Refer to [Appendix A](#) for configuration details.

For the benchmark of symmetric crypto operation, the small packet offload has been explicitly enabled to show the raw performance of the hardware accelerator. When this option is disabled (default), the performance for small packets are almost on par with software. As of OpenSSL-1.1.0, the pipeline feature was not enabled in the standard release. In conjunction with async, this feature will further increase bulk cipher performance.

4.4 Application-Level Benchmark (NGINX-1.10 + OpenSSL-1.1.0)

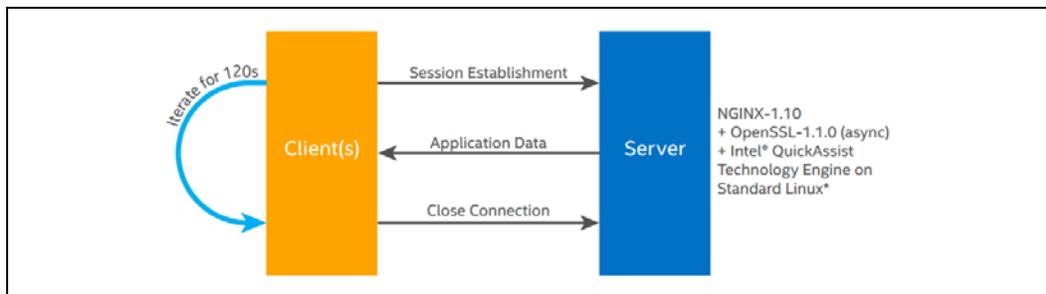
SSL/TLS is predominantly used in client-server based applications to provide security for communication of data. From a security protocol perspective, TLS's ability to secure communications between two TCP ports is one of its primary advantages. An individual port typically translates to an individual application on a client system, providing isolation between the many applications that could be potential attack points for the client.

This client-centered view is important. As the benchmark metrics will show, client metrics are the primary performance driver. For a client, the ability to connect to many services and transfer data seamlessly is key. On the server side, this translates into the number of new connections per second a server can create. This viewpoint identifies the key metric to drive analysis with the number of SSL/TLS handshakes per second for a SSL/TLS server.

4.5 Benchmark Topology

For the measurement of SSL/TLS performance, a web server was analyzed using NGINX* as the management application. The topology for the benchmark is shown in the following figure.

Figure 5. Client-Server Benchmark Overview



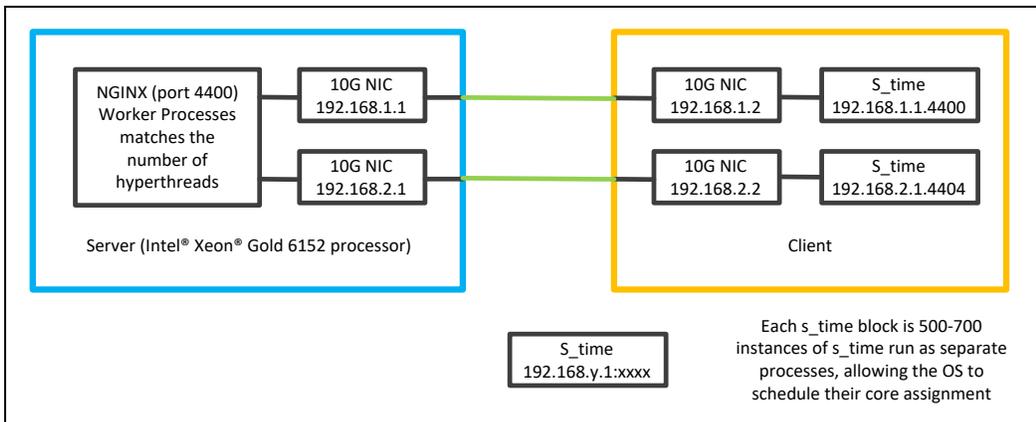
Note: For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

In this configuration, a large number of clients are connected to a single server system, with each client running in a loop fetching data from the server. Once each client completes its operation, it will loop and initiate the same operation again, running for a predetermined amount of time.



The following figure shows the physical topology for the Web Server benchmark used for TLS handshake measurements. The client issues requests using `s_time`, a utility provided with OpenSSL, which is forked to provide 500-700 instances of `s_time` running simultaneously to scale to full performance. Each instance runs as a separate process and targets a specific TCP port on the server. The `s_time` command line used is used in the following form: `./openssl-1.0.2d/apps/openssl s_time -connect 192.168.1.1:4400 -new -nbio -time 200 -cipher <cipher suite e.g. AES128-SHA>`

Figure 6. Web-Server Physical Topology



Note: Refer to [Appendix A](#) for configuration details.

On the server side, the NGINX* configuration file (along with the client request) coordinate the cipher suites and key negotiation algorithms to use in each test. For these benchmarks, the following combinations were used:

Table 7. TLS Benchmark Test Parameters

Application Data	0 byte file size
Key Negotiation Algorithm	RSA-2K ECDHE-RSA-2K (P256) ECDHE-ECDSA (P256)
Protocol	TLS v1.2
Cipher Suite	AES_128_CBC_HMAC_SHA1



Figure 7. RSA-2K Connections per Second (NGINX-1.10 + OpenSSL-1.1.0)

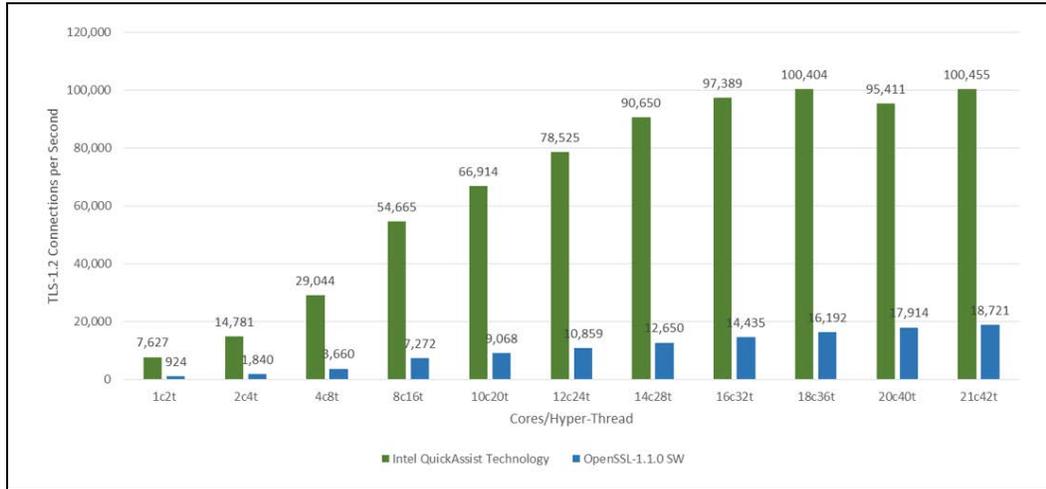


Figure 8. ECDHE-RSA-2K Connections per Second (NGINX-1.10 + OpenSSL-1.1.0)

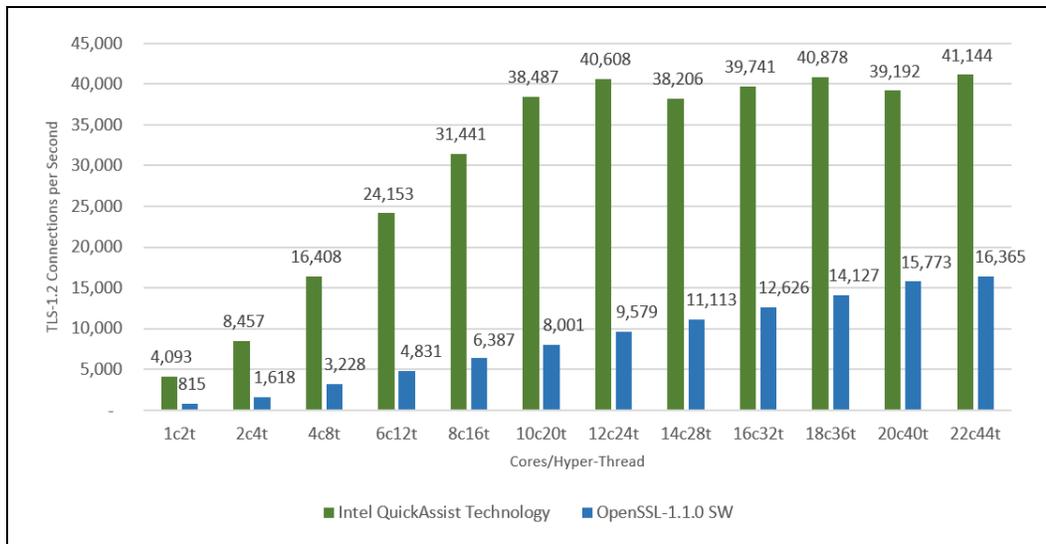




Figure 9. ECDHE-ECDSA Connections per Second (NGINX-1.10 + OpenSSL-1.1.0)

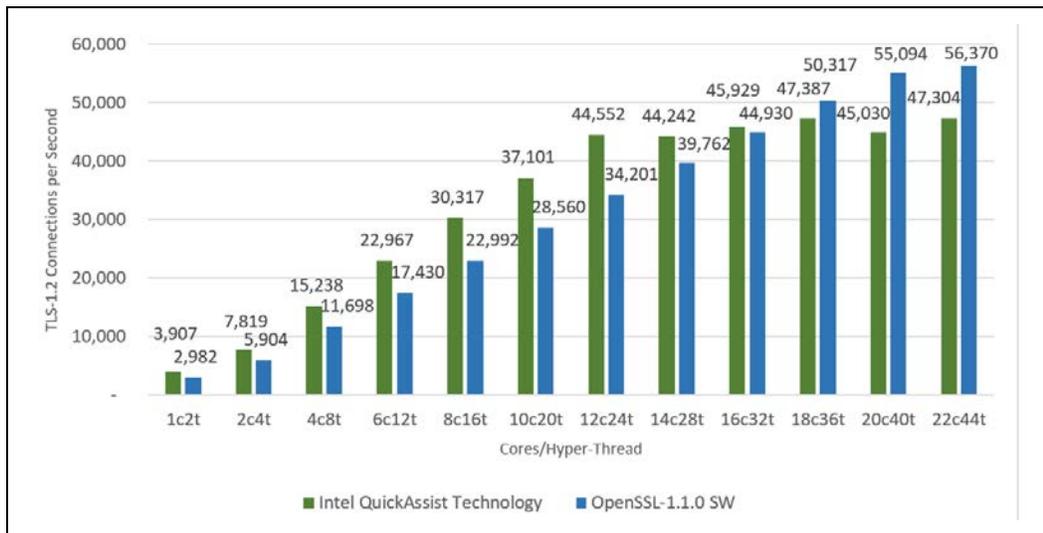


Figure 7, Figure 8, and Figure 9 represent measurements taken with the same benchmark and stack, using Intel® QAT with OpenSSL-1.1.0 asynchronous features and OpenSSL-1.1.0's default software implementations for the three most popular key negotiation algorithms (ECDHE-RSA2K (P256), ECDHE-ECDSA (P256), and RSA-2K). This benchmark is scaled by limiting the resources available to NGINX* to a specified number of cores and hyperthreads. If we look in more detail at the one core two hyper thread (1C/2T) measurement for RSA-2K TLS-1.2:

- OpenSSL-1.1.0 Software: 964 CPS (connections per second)
- Intel® QuickAssist Technology Engine: 5,507 CPS

In conjunction with the new asynchronous features added to OpenSSL-1.1.0, Intel® QAT is able to achieve a performance gain of approximately 825% performance compared to the standard software implementation. This performance increase is measured by issuing a sufficient number of client TLS connections to drive the server processor for the chosen core configuration to greater-than-90% utilization. This method is then extended to additional core configurations until reaching the limit of Intel® QAT's ability to calculate the cryptographic operations being targeted. The limits for the Intel® Communications Chipset 8950 are as follows:

- RSA-2K: 102 K decryptions per second
- ECDHE+RSA-2K: 42.03 K operations per second
- ECDHE+ECDSA (P256): 48.33 K operations per second

Intel® QuickAssist Technology + OpenSSL-1.1.0 asynchronous features deliver a gain over software (up to the device limits) of:

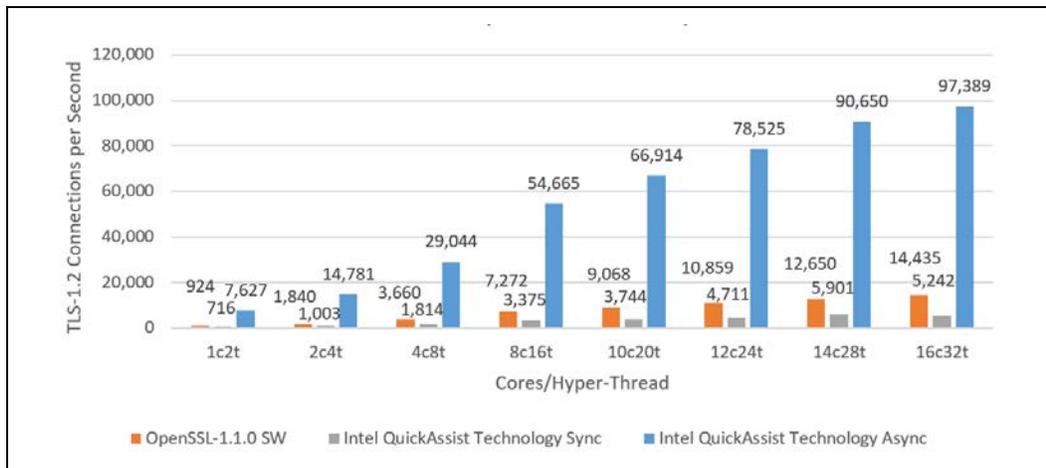
- RSA-2K ~5.5 times
- ECDHE-RSA-2K ~5 times
- ECDHE-ECDSA ~1.3 times

Note: Further improvements are currently being developed to increase performance gains when using ECDHE-ECDSA.



The core to hyperthread pairing for asynchronous measurements scales well for all algorithms, meaning as more cores are added, the increase in connections per second trends linearly. For RSA-2K, the limit of the hardware accelerator is reached using around 18 cores/36 hyper threads (which translates to 360 NGINX* worker processes), while for ECDHE-RSA-2K and ECDHE-ECDSA, they scale well up to the hardware limits.

Figure 10. Comparison of Synchronous versus Asynchronous Infrastructure



These three modes can be easily configured using the NGINX* patch, which allows you to enable and disable the QAT_ engine using the NGINX* config directive `ssl_engine qat;`. Similarly, `synch/asynch` can be toggled using the directive `ssl_asynch on`. The benchmark is run with the same mapping of NGINX* worker processes to cores/hyper threads as previously mentioned.

As shown by the results in [Figure 10](#), there is a clear advantage to using the asynchronous model when a separate processing entity is present for cryptographic calculations within the system. It allows for efficient parallel processing of the workloads and better utilization of the available hardware. It should be noted that while similar performance results can be achieved by parallelizing with a high level context mechanism such as threads or processes, those methods consume significantly more processor cycles to achieve the same results as the asynchronous model.



5 Conclusion

The OpenSSL 1.1.0 release provides many new features. This work has focused on the asynchronous capabilities, which demonstrate a significant performance gain when utilizing a high-performing asynchronous engine (Intel® QuickAssist Technology C627) with the ASYNC_JOB infrastructure to efficiently manage the processing flow. For asymmetric cryptographic algorithms such as RSA-2K, there is a demonstrable performance gain of 7.5 times versus software with the same number of cores, while at the algorithm level the gain shows a more dramatic performance increase of 36 times.

While these levels of performance may already be achievable with bespoke SSL/TLS stacks, this is the first introduction to the mainline of a popular SSL/TLS stack enabling these levels. As a standardized interface in OpenSSL, it lays the infrastructure for many applications to adopt these features and opens a significant potential to increase SSL/TLS more generally.

§



Appendix A Platform Details

Hardware		
Motherboard		Intel® Purley
Processor	Product	Intel® Xeon® Gold 6152 processor @ 2.10 GHz
	Speed (MHz)	2.10 GHz
	No of processor	22 cores /33 threads per socket
	Stepping	H0
	Technology	14 nm
	Supported Instructions Sets	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch epb intel_pt tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx avx512f rdseed adx smap clflushopt clwb avx512cd xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req
	Level 1 Data Cache	32 KB
	Level 1 Instruction Cache	32 KB
	L2 Cache	1024 KB
	LLC Cache	30976 KB
Memory	Vendor	Kingston
	Type	DDR4-2400
	Part Number	KVR24R170/16
	Size	16384 MB
	Channel	6
BIOS	Vendor	Intel
	Version	PLYDCRB1.86B.0114.R11.1612211902
	Build Date	21-Dec-16
Compiler Versions	GCC Version	5.4.0
	Linker Version (LD)	2.26.1



	Assembler Version (AS)	2.26.1
Software		
OS	Vendor	Ubuntu 16.04.1 LTS
	Kernel Version	4.4.0-21
Benchmark Software	Glibc-2.24 <u>CBC Data Collected on:</u> NGINX-1.9.6 OpenSSL-1.0.1u QATL1.0.0-15 <u>GCM and PKE Data Collected on:</u> NGINX-1.10-3 OpenSSL-1.1.0e QATL.1.0.0-15 QAT Engine v0.5.24	

Table 8. References

Item #	Description
[1]	"QAT_engine [github]" https://github.com/01org/QAT_Engine .
[2]	"ASYNC_start_job" https://www.openssl.org/docs/man1.1.0/crypto/ASYNC_start_job.html .
[3]	"rfc5246," https://www.ietf.org/rfc/rfc5246.txt .
[4]	"Pseudo Random Function (API addition)" https://github.com/openssl/openssl/commit/1eff3485b63f84956b5f212aa4d853783bf6c8b5 .
[5]	"PRF integration into libssl" https://github.com/fernandmorausky/openssl/commit/b7d60e7662f903fc2e5a137bf1fce9a6b431776a .
[6]	"PRF EVP API man page" https://www.openssl.org/docs/manmaster/man3/EVP_PKEY_CTX_set_tls1_prf_md.html .
[7]	"ASYNC_WAIT_CTX" https://www.openssl.org/docs/manmaster/man3/EVP_PKEY_CTX_set_tls1_prf_md.html .
[8]	"SSL_get_error" https://www.openssl.org/docs/man1.1.0/ssl/SSL_get_error.html .
[9]	"SSL_get_all_async_fds" https://www.openssl.org/docs/man1.1.0/ssl/SSL_get_all_async_fds.html .
[10]	"SSL_CTX_set_max_pipelines" https://www.openssl.org/docs/man1.1.0/ssl/SSL_CTX_set_max_pipelines.html .



Item #	Description
[11]	"SSL_CTX_set_split_send_fragment" https://www.openssl.org/docs/man1.1.0/ssl/SSL_CTX_set_split_send_fragment.html .
[12]	"Intel QuickAssist Technology engine build options" https://github.com/01org/QAT_Engine#intel-quickassist-technology-openssl-engine-build-options .
[13]	"OpenSSL configuration file" https://github.com/01org/QAT_Engine#using-the-openssl-configuration-file-to-loadinitialize-engines .
[14]	"ENGINE_set_default" https://www.openssl.org/docs/man1.1.0/crypto/ENGINE_set_default.html .
[15]	"QAT_engine specific messages" https://github.com/01org/QAT_Engine#intel-quickassist-technology-openssl-engine-specific-messages .
[16]	"SSL_set_mode" https://www.openssl.org/docs/man1.1.0/ssl/SSL_set_mode.html .
[17]	"Async job pool" https://github.com/openssl/openssl/blob/OpenSSL_1_1_0-stable/ssl/ssl_lib.c .
[18]	"Epoll openssl speed" https://github.com/openssl/openssl/blob/OpenSSL_1_1_0-stable/apps/speed.c .
[19]	"Dummy Async Engine" https://github.com/openssl/openssl/blob/OpenSSL_1_1_0-stable/engines/e_dasync.c .

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information, visit <http://www.intel.com/performance>.