An Empirical Study of Cryptographic Libraries for MPI Communications

Abu Naser
Department of Computer Science
Florida State University
Tallahassee, FL, USA
naser@cs.fsu.edu

Mohsen Gavahi
Department of Computer Science
Florida State University
Tallahassee, FL, USA
gavahi@cs.fsu.edu

Cong Wu
Department of Computer Science
Florida State University
Tallahassee, FL, USA
wu@cs.fsu.edu

Viet Tung Hoang
Department of Computer Science
Florida State University
Tallahassee, USA
tvhoang@cs.fsu.edu

Zhi Wang
Department of Computer Science
Florida State University
Tallahassee, FL, USA
zwang@cs.fsu.edu

Xin Yuan
Department of Computer Science
Florida State University
Tallahassee, FL, USA
xyuan@cs.fsu.edu

***Abstract*—As High Performance Computing (HPC) applications with data security requirements are increasingly moving to execute in the public cloud, there is a demand that the cloud infrastructure for HPC should support privacy and integrity. Incorporating privacy and integrity mechanisms in the communication infrastructure of today's public cloud is challenging because recent advances in the networking infrastructure in data centers have shifted the communication bottleneck from the network links to the network end points and because encryption is computationally intensive.**

In this work, we consider incorporating encryption to support privacy and integrity in the Message Passing Interface (MPI) library, which is widely used in HPC applications. We empirically study four contemporary cryptographic libraries, OpenSSL, BoringSSL, Libsodium, and CryptoPP using micro-benchmarks and NAS parallel benchmarks to evaluate their overheads for encrypting MPI messages on two different networking technologies, 10Gbps Ethernet and 40Gbps InfiniBand. The results indicate that (1) the performance differs drastically across cryptographic libraries, and (2) effectively supporting privacy and integrity in MPI communications on high speed data center networks is challenging—even with the most efficient cryptographic library, encryption can still introduce very significant overheads in some scenarios such as a single MPI communication operation on InfiniBand, but (3) the overall overhead may not be prohibitive for practical uses since there can be multiple concurrent communications.

***Index Terms*—MPI, encrypted communication, benchmark.**

This material is based upon work supported by the National Science Foundation under Grants CICI-1738912, CRI-1822737, CNS-1453020, and CRII-1755539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. This work used the XSEDE Bridges resource at the Pittsburgh Supercomputing Center (PSC) through allocation TG-ECS190004.

978-1-7281-4734-5/19/$31.00 ©2019 IEEE

I. Introduction

High performance computing (HPC) applications often process highly-sensitive data, such as medical, financial, and engineering documents. As more and more HPC applications are executing in the public cloud, there is a pressing need that the cloud infrastructure for HPC should provide privacy and integrity. One important component of the cloud infrastructure for HPC is the Message Passing Interface (MPI) library, the de facto and vastly popular communication library for message passing applications. For MPI applications to run with strong security guarantees in the public cloud, privacy and integrity mechanisms based on modern cryptographic theories and algorithms must be incorporated in the MPI library.

Unfortunately, the existing efforts to retrofit MPI libraries with encryption contain severe security flaws. For example, ES-MPICH2 [1], the first such MPI library, uses the weak ECB (Electronic Codebook) mode of operation that has known vulnerabilities [2, page 89]. In addition, no existing encrypted MPI libraries provide meaningful data integrity, meaning that data could potentially be modified without being detected. Consequently, it is urgent to revisit the problem by applying the state-of-art cryptographic theory and practice to properly encrypt MPI communications.

In recent years, significant efforts have been put to improve the security, usability, and performance of cryptographic libraries. Popular cryptographic libraries, including OpenSSL [3], BoringSSL [4], Libsodium [5] and CryptoPP [6], all received intensive security review, and some (OpenSSL and CryptoPP) even passed the Federal Information Processing Standards (FIPS) 140-2 validation. In addition, recent processors from all major CPU vendors have introduced hardware support to speed up cryptographic operations (e.g., Intel AES-NI instructions to accelerate the AES algorithm, or the x86 CLMUL instruction set to improve the speed of finite-field multiplications). All of the popular cryptographic

libraries now support hardware-accelerated cryptographic operations. Nevertheless, those libraries have different usability, functionality, and performance; and it is unclear which library is the best option for different types of MPI communication paradigms. We thus need to understand how the selection of cryptographic libraries can affect the security and performance of common MPI applications.

The advances in the networking infrastructure in data centers have shifted the communication bottleneck from the network links to the network end-points. As such, when incorporating security mechanisms in the MPI library, the additional computation in the cryptographic operations likely will introduce significant overheads to MPI communications, detrimental to the performance. It is thus critical to understand the overheads introduced in the cryptographic operations and to find the most efficient cryptographic library for MPI communications.

In this work, we develop encrypted MPI libraries that are built on top of four cryptographic libraries OpenSSL, BoringSSL, Libsodium, and CryptoPP. Using these libraries, we empirically evaluate the performance of encrypted MPI communications with micro benchmarks and NAS parallel benchmarks [7] on two networking technologies, 10Gbps Ethernet and 40Gbps InfiniBand QDR. The main conclusions include the following:

- Different cryptographic libraries result in very different overheads. Specifically, OpenSSL and BoringSSL are on par with each other; and their performance is much higher than that of Libsodium and CryptoPP, on both Ethernet and InfiniBand.
- For individual communication, encrypted MPI introduces relative small overhead for small messages and large overhead for large messages. For example, on the 10Gbps Ethernet, even for BoringSSL, encrypted MPI under the AES-GCM encryption scheme [8] introduces 5.9% overhead for 256-byte messages and 78.3% for 2MB messages in the ping-pong test. On the 40Gbps InfiniBand, BoringSSL introduces 80.9% overhead for 256-byte messages and 215.2% overhead for 2MB messages in the ping-pong test. This calls for developing new techniques to optimize the combination of encryption and MPI communications.
- For more practical scenarios, the cryptographic overhead is not as significant. On average, BoringSSL only introduces 12.75% overhead on Ethernet and 17.93% overhead on InfiniBand for NAS parallel benchmarks (class C running on 64 processes and 8 nodes).

II. Related work

There have been a few proposed systems for adding encryption to MPI libraries, and some have even been implemented [1], [9]–[12]. Existing systems, however, suffer from notable security vulnerabilities, as we will elaborate below.

First, privacy—the main goal of those systems—is seriously flawed because of the insecure crypto algorithms or the misuse of crypto algorithms. For example, ES-MPICH2 [1] is the first MPI library that integrates encryption to MPI communication, but its implementation is based on a weak encryption scheme, the Electronic Codebook (ECB) mode of operation. While ECB is still included in several standards, such as NIST SP 800-38A, ANSI X3.106, and ISO 8732, it has been known to be insecure [2, page 89]. For another example of an insecure choice of encryption, consider the system VAN-MPICH2 [11] that relies on one-time pads for encryption. It however implements one-time pads as substrings of a big key K. Thus when encrypting many large messages, it is likely that there are two messages M_1 and M_2 whose one-time pads L_1 and L_2 are overlapping substrings of K, say the last 8 KB of L_1 is also the first 8 KB of L_2. In that case, one can obtain the xor of X_1 and X_2, where X_1 is the last 8 KB of M_1 and X_2 is the first 8 KB of M_2. If X_1 and X_2 are English texts there are known methods to recover them from their xor value [13].

Next, no existing system provides meaningful data integrity. Some do suggest that integrity may be added via digital signatures [1], [11], but this is impractical because all existing digital signature schemes are expensive. Some consider encrypting each message together with a checksum (obtained via a cryptographic hash function such as SHA-2) [10], but this approach does not provide integrity if one uses classical encryption schemes such as the Cipher Block Chaining (CBC) mode of encryption [14]. Others believe that encrypting data via the ECB mode also provides integrity [1], [9], but it is well-known that classical encryption schemes such as ECB or CBC provide no integrity at all [2, page 109].

We note that the insecurity of the systems above has never been realized in the literature. MPI communication therefore is in dire need of strong encryption that provides both privacy and integrity. In addition, in recent years, hardware support for efficient cryptographic operations, such as Intel's AES-NI instructions, has become ubiquitous. These advances are fully exploited by modern cryptographic libraries to improve encryption speed. Yet there is currently a lack of understanding of how these libraries perform in the MPI environment. Our paper fills this gap, giving (i) the first implementation that properly encrypts MPI communication to provide genuine privacy and integrity, and (ii) a systematic benchmarking to investigate the overheads of modern cryptographic libraries for MPI communication on contemporary clusters. Unlike prior work with insecure, ad hoc encryption schemes, our implementation is based on the Galois-Counter Mode (GCM) that *provably* delivers both privacy and integrity [15].

III. Background

A. Encryption Schemes

A (symmetric) encryption scheme is a triple of algorithms $(\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$. Initially, the sender and receiver somehow manage to share a secret key K that is randomly generated by $\texttt{Gen}$. Each time the sender wants to send a message M to the receiver, she would encrypt $C \leftarrow \texttt{Enc}(K, M)$, and then send the ciphertext C in the clear. The receiver, upon receiving C, will decrypt $M \leftarrow \texttt{Dec}(K, C)$. An encryption

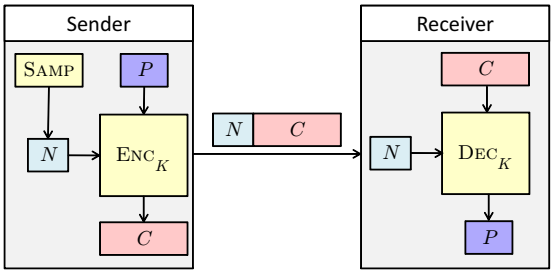

Fig. 1: Encrypted communication with AES-GCM.

scheme is commonly built on top of a blockcipher (such as AES and 3DES).

Standard documents, such as NIST SP 800-38A [16] and 800-38D [8], specify several modes of encryption. Many of them, such as Electronic Codebook (ECB), Cipher Block Chaining (CBC), Counter (CTR), Galois/Counter Mode (GCM), and Counter with CBC-MAC (CCM), are well-known and widely used. However, these schemes are not equal in security and ease of correct use. The ECB mode, for example, is insecure [2]. CBC and CTR modes provide only privacy, meaning that the adversary cannot even distinguish ciphertexts of its chosen messages with those of uniformly random messages of the same length. They however do not provide data integrity in the sense that the adversary cannot modify ciphertexts without detection.[1] Among the standardized encryption schemes, only GCM and CCM satisfy both privacy and integrity, but GCM is the faster one [17]. Therefore, in this paper, we will focus on GCM; one does not need to know technical details of GCM to understand our paper.

According to NIST SP 800-38D, the blockcipher for GCM must be AES, and correspondingly, the key length is either $128, 192,$ or 256 bits. The longer key length means better security against brute-force attacks, but also slower speed. In this paper, we consider both 128-bit key (the most efficient version) and 256-bit key (the most secure one). AES-GCM is a highly efficient scheme [17], provably meeting both privacy and integrity [15]. Due to its strength, AES-GCM appears in several network protocols, such as SSH, IPSec, and TLS.

Syntactically, AES-GCM is a *nonce-based* encryption scheme, meaning that to encrypt plaintext P, one needs to additionally provide a *nonce* N, i.e., a public value that must appear at most once per key. The same nonce N is required for decryption, and thus the sender needs to send both the nonce N and the ciphertext C to the receiver. See Fig. 1 for an illustration of the encrypted communication via GCM. In AES-GCM, nonces are 12-byte long, and one often implements them via a counter, or pick them uniformly at random. In addition, each ciphertext is 16-byte longer than the corresponding plaintext, as it includes a 16-byte tag to determine whether the ciphertext is valid.

[1] Actually, the adversary can still replace a ciphertext with a prior one; this is known as *replay attack*. Here we do not consider such attacks.

B. Cryptographic Libraries

In our implementation, we consider the following cryptographic libraries: OpenSSL [3], BoringSSL [4], Libsodium [5], and CryptoPP [6]. They are all in the public domain, are widely used, and have received substantial scrutiny from the security community.

OpenSSL is one of the most popular cryptographic libraries, providing a widely used implementation of the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. Due to its importance, there has been a long line of work in checking the security of OpenSSL, resulting in the discovery of several important vulnerabilities, such as the notorious Heartbleed bug [18]. As a popular commercial-grade toolkit, OpenSSL is used by many systems. BoringSSL is Google's fork of OpenSSL, providing the SSL library in Chrome/Chromium and Android OS.

Libsodium is a well-known cryptographic library that aims for security and ease of correct use. It provides many benefits such as portability, cross-compilability, and API-compatibility, and supports bindings for all common programming languages. As a result, Libsodium has been used in a number of applications, such as the cryptocurrency Zcash and Facebook's OpenR (a distributed platform for building autonomic network functions). It however only supports AES-GCM with 256-bit keys.

CryptoPP is another popular open-source cryptographic library for C++. It is widely adopted in both academic and commercial usage, including WinSSHD (an SSH server for Windows), Steam (a digital distribution platform purchasing and playing video games), and Microsoft SharePoint Workspace (a document collaboration software).

IV. MPI WITH ENCRYPTED COMMUNICATION

We developed two MPI libraries whose communication is encrypted via AES-GCM (for both 128-bit and 256-bit keys); one library is based on MPICH-3.2.1 for Ethernet and the other on MVAPICH2-2.3 for InfiniBand. Specifically, encryption is added to the following MPI routines:

- Point-to-point: `MPI_Send`, `MPI_Recv`, `MPI_ISend`, `MPI_IRecv`, `MPI_Wait`, and `MPI_Waitall`.
- Collective: `MPI_Allgather`, `MPI_Alltoall`, `MPI_Alltoallv`, and `MPI_Bcast`.

The underlying cryptographic library is user-selectable among OpenSSL, BoringSSL, Libsodium, and CryptoPP. With encryption incorporated at the MPI layer, our prototypes can run on top of any underlying network. As our main focus of this work is to benchmark the performance of encrypted MPI libraries, we did not implement a key distribution mechanism; this is left as a future work. In our experiments, the encryption key was hardcoded in the source code.

To illustrate the high-level ideas of our implementation, consider the pseudocode of our `Encrypted_Alltoall` routine in Algorithm 1. Within this code, we use `RAND_bytes`(s) to denote the sampling of a uniformly random s-byte string, and $X \,\|\, Y$ for the concatenation of two strings X and Y. The

Input: Two arrays $sendbuf$ and $recvbuf$, each of $n+1$ elements that are ℓ-byte long.
Parameter: A key K
`/* Create ciphertext buffers */`
Initialize two arrays $enc\_sendbuf$ and $enc\_recvbuf$, each of $n+1$ elements that are $(\ell+28)$-byte long.
for $i \leftarrow 0$ **to** n **do**
  `/* Get a random 12-byte nonce */`
  $N_i \leftarrow$ `RAND_bytes`(12)
  `/* Encrypt via AES-GCM */`
  $C_i \leftarrow$ `Enc`$(K, N_i, sendbuf[i])$
  `/* Concatenate nonce and ctx */`
  $enc\_sendbuf[i] \leftarrow N_i \,\|\, C_i$
end
`MPI_Alltoall`$(enc\_sendbuf, enc\_recvbuf)$
for $i \leftarrow 0$ **to** n **do**
  `/* Parse to nonce/ciphertext */`
  $N_i^* \,\|\, C_i^* \leftarrow enc\_recvbuf[i]$
  `/* Decrypt via AES-GCM */`
  $recvbuf[i] \leftarrow$ `Dec`(K, N_i^*, C_i^*)
end

Algorithm 1: `Encrypted_Alltoall` routine.

encryption and decryption routines of AES-GCM are `Enc` and `Dec`, respectively. Intuitively, the ordinary `MPI_Alltoall` is used to send/receive just ciphertexts and their corresponding nonces. That is, one would need to encrypt the sending messages before calling `MPI_Alltoall`—each message with a fresh random nonce, and then decrypt the receiving ciphertexts. If a sending message is ℓ-byte long then the corresponding data that `MPI_Alltoall` sends is $(\ell+28)$-byte long, since it consists of (i) a 12-byte nonce, and (ii) a ciphertext that is 16-byte longer than its plaintext.

We note that although the pseudocode above seems straightforward, in an actual implementation, there are some low-level subtleties when one has to deal with non-blocking communication. For example, for `Encrypted_IRecv`, our implementation performs decryption inside `MPI_Wait` to ensure the non-blocking property.

V. Experiments

We empirically evaluated the performance of our encrypted MPI libraries to (i) understand the encryption overheads in MPI settings, and (ii) determine the best cryptographic library to use with MPI. Below, we will first describe the system of our experiments, the benchmarks, and our methodology. Later, in Section V-A and Section V-B, we will report the experiment results on Ethernet and Infiniband respectively.

System setup. The experiments were performed on a cluster with the following configuration. The processors are Intel Xeon E5-2620 v4 with the base frequency of 2.10 GHz. Each node has 8 cores and 64GB DDR4 RAM and runs CentOS 7.6. Each node is equipped with two types of network interface cards: a 10 Gigabit Ethernet card (Intel 82599ES SFI/SFP+) and a 40 Gigabit InfiniBand QDR one (Mellanox MT25408A0-FCC-QI ConnectX). Allocated nodes were chosen manually. For each experiment, the same node allocation was repeated for all measurements. All ping-pong results use two processes on different nodes.

We implemented our prototypes on top of MPICH-3.2.1 (for Ethernet) and MVAPICH2-2.3 (for Infiniband). The baseline and our encrypted MPI libraries were compiled with the standard set of MPICH and MVAPICH compilation flags and optimization level O2. In addition, we compiled all the cryptographic libraries (OpenSSL 1.1.1, BoringSSL, CryptoPP 7.0, and Libsodium 1.0.16) separately using their default settings and linked them with MPI libraries during the linking phase of MPICH and MVAPICH.

Benchmarks. We consider the following suites of benchmarks:

- Encryption-decryption: The encryption-decryption benchmark measures the encryption and decryption performance. For each data size, it measures the time for performing 500,000 times the simple encryption and then decryption of the data using a single thread.
- Ping-pong: This benchmark measures the uni-directional throughput when two processes communicate back and forth repeatedly using blocking send and receive. We ran several experiments, each corresponding to a particular message size within the range from 1B to 2MB. In each experiment measurement, the two processes send messages of the designated size back and forth 10,000 times if the message size is less than 1MB, and 1,000 times otherwise. For encrypted communication, each message results in an additional 28-byte overhead, as we need to send a 12-byte nonce and a 16-byte tag per ciphertext. Those bytes are excluded in the throughput calculation.
- OSU micro-benchmark 5.4.4 [19]: We used the Multiple Pair Bandwidth Test benchmark in OSU suite to measure aggregate uni-directional throughput when multiple senders in one node communicate with their corresponding receivers in another node, via non-blocking send and receive. In each experiment measurement, the sender iterates 100 times; in each iteration, it sends 64 messages of the designated size to the receiver and wait for the replies before moving to the next iteration. Again, we excluded the 28-byte overhead per message in calculating the throughput.

  We also used OSU suite to measure performance of collective communication routines. Each experiment measurement consists of 100 iterations.
- NAS parallel benchmarks [7]: To measure performance of (encrypted) MPI in applications, we used the BT, CG, FT, IS, LU, MG, and SP in the NAS parallel benchmarks; all experiments used Class C size.

BENCHMARK METHODOLOGY. For ping-pong, OSU benchmarks, and NAS benchmarks, we first ran each experiment at least 20 times, up to 100 times, until the standard deviation was within 5% of the arithmetic mean. If after 100 measurements, the standard deviation was still too big then we would keep running the experiment until the 99% confidence interval was within 5% of the mean. The variability for encryption and decryption is much smaller. Hence, each result for the encryption-decryption benchmark is obtained by running the benchmark at least 5 times until the standard deviation was within 5% of the arithmetic mean.

To evaluate the scalability of our implementation, we used four different settings (e.g. 4 rank/4 node, 16 rank/4 node, 16 rank/8 node and 64 rank/8 node) for OSU and NAS benchmarks.

WHAT WE REPORT. In our experiments, BoringSSL and OpenSSL delivered very similar performance. This is not surprising, since BoringSSL is a fork of OpenSSL. In addition, the benchmarks yielded the same trends for both 128-bit and 256-bit keys. We therefore only report the results of BoringSSL (256-bit key), Libsodium, and CryptoPP (256-bit key).

A. *Ethernet Results*

ENCRYPTION-DECRYPTION. Before we get into the details of the communication benchmark results, it is instructive to understand the performance of AES-GCM, since it helps us to have a better understanding of the performance of the encrypted MPI libraries. The average throughputs of AES-GCM-256 with different data sizes are shown in Fig. 2. It is clear that different encryption libraries have very different encryption and decryption performance. There are two ways to interpret the results here. First, one can view this as the convergence of the ping-pong performance when the network speed becomes much faster than encryption and descryption. Also, since for AES-GCM, the encryption and decryption speed is roughly the same, the reported performance here is a half of the encryption throughput (that is also decryption throughput). Thus we can predict that for most experiments, among the three encrypted MPI libraries, BoringSSL is the best, and then Libsodium, and finally CryptoPP.

PING-PONG. The ping-pong performance of the baseline and the encrypted MPI libraries is shown in Table I for small messages, and illustrated in Fig. 3 for medium and large messages. For 1KB messages, BoringSSL appears to slightly outperform the baseline, but recall that we are reporting the mean values with 5% deviation, so this only means that BoringSSL has very close performance to the baseline.

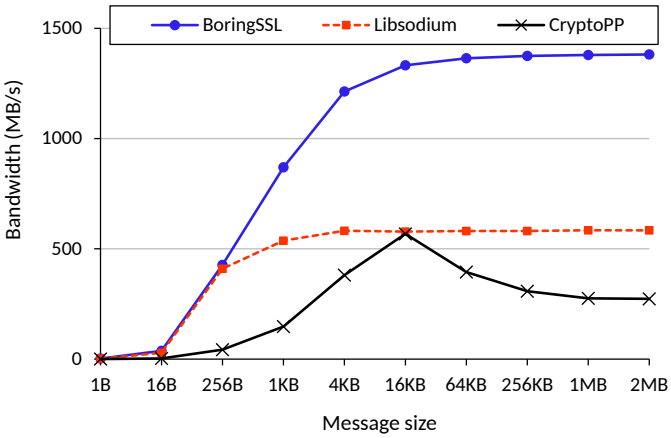

Fig. 2: Encryption-decryption throughput of AES-GCM-256, compiled with the gcc 4.8.5.

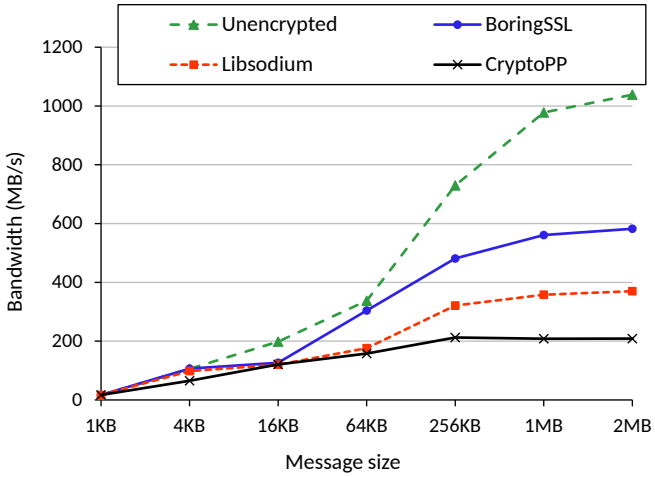

Fig. 3: Average unidirectional ping-pong throughput with 256-bit encryption key on Ethernet, for medium and large messages.

	1B	16B	256B	1KB
Unencrypted	0.050	0.83	7.01	17.03
BoringSSL	0.045	0.78	6.62	17.05
Libsodium	0.046	0.79	6.62	17.02
CryptoPP	0.029	0.48	6.85	17.02

TABLE I: Average unidirectional ping-pong throughput (MB/s) for small messages, with 256-bit encryption key on Ethernet.

For large messages, say 2MB ones, encrypted MPI libraries have poor performance compared to the baseline: even the fastest BoringSSL yields 78.3% overhead, and CryptoPP's overhead is much worse, nearly 400%. These performance results can be explained as follows.

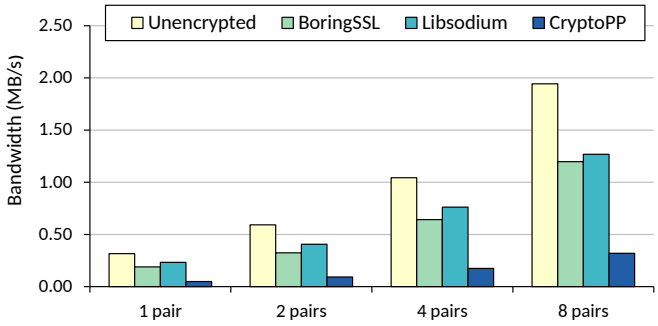

Fig. 4: OSU Multiple-Pair average throughput for 1B-messages on Ethernet.

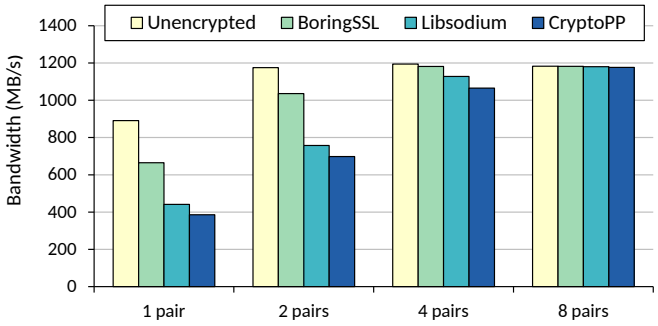

Fig. 5: OSU Multiple-Pair average throughput for 16KB-messages on Ethernet.

- The running time of an encrypted MPI library consists of (i) the encryption-decryption cost, and (ii) the underlying MPI communications, which roughly corresponds to the baseline performance.
- For BoringSSL, on 2MB messages, the encryption-decryption throughput of AES-GCM-256 (1381 MB/s) is about 1.32 times that of the ping-pong throughput of the baseline (1038 MB/s). Estimatedly, BoringSSL's ping-pong time would be roughly $\frac{1+1.32}{1.32} \approx 1.76$ times slower than that of the baseline. This is consistent with the reported 78.3% overhead above.
- For CryptoPP, on 2MB messages, the encryption-decryption throughput of AES-GCM-256 (273 MB/s) much worse, just around 26% of the ping-pong performance of the baseline (1038 MB/s). One thus can estimate that CryptoPP's ping-pong time would be about $\frac{1+0.26}{0.26} \approx 4.84$ times slower than that of the baseline. This is again consistent with the reported 400% overhead above.

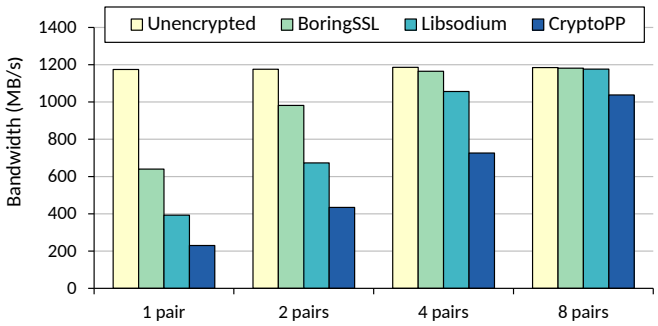

Fig. 6: OSU Multiple-Pair average throughput for 2MB-messages on Ethernet.

For small messages, encrypted MPI libraries often perform reasonably well, since the encryption-decryption throughput of AES-GCM-256 is quite higher than the ping-pong throughput of the baseline. For example, for 256-byte messages, the encryption-decryption throughput of Libsodium is 409.67 MB/s, much higher than the 7.01 MB/s baseline ping-pong throughput. Consequently, Libsodium has just 5.89% overhead for 256-byte messages.

OSU MULTIPLE-PAIR BANDWIDTH. The Multiple-Pair performance of the baseline and the encrypted MPI libraries, for 1B, 16KB, and 2MB messages, is shown in Figures 4, 5, and 6, respectively.

For medium and large messages, as the number of pairs increases, the relative performance of the encrypted MPI libraries becomes much better, because (i) the network bandwidth remains the same, yet the computational power doubles, and (ii) encryption/decryption can overlap with MPI communications. When there is just a single pair, even BoringSSL cannot encrypt fast enough to keep up with the network speed. However, when there are 8 pairs, even CryptoPP can reach the baseline performance, for 16KB messages. These results suggest that (1) the overhead for a single communication flow may be significant, but (2) in modern multi-core machines, when multiple flows happen concurrently, the performance of encrypted MPI libraries may be on par with the baseline.

For small messages, the situation is different, because the network bandwidth is not fully used. As shown in Fig. 4, the baseline throughput keeps increasing as the number of pairs increases. In contrast, in Figures 5 and 6, for medium and large messages, the baseline throughput is saturated when there are just two pairs of senders and receivers. Consequently, even when there are 8 pairs, BoringSSL still incurs 61.67% overhead, and CryptoPP is far worse, resulting in 506.25% overhead.

COLLECTIVE COMMUNICATION. The average running time of `Encrypted_Bcast` and `Encrypted_Alltoall`, for the 64-rank and 8-node setting, is shown in Tables II and III, respectively.

	1B	16KB	4MB
Unencrypted	31.15	231.75	9,594.75
BoringSSL	37.15	246.17	13,892.74
Libsodium	35.54	264.37	18,322.19
CryptoPP	54.97	278.65	29,301.96

TABLE II: Average timing of `Encrypted_Bcast` (μs), with 256-bit encryption key on Ethernet.

	1B	16KB	4MB
Unencrypted	159.13	6,562.82	1,966,299.47
BoringSSL	329.60	7,691.08	2,210,546.32
Libsodium	452.76	8,937.74	2,535,104.93
CryptoPP	1,221.98	9,462.90	3,297,402.93

TABLE III: Average timing of `Encrypted_Alltoall` (μs), with 256-bit encryption key on Ethernet.

To understand the performance of `Encrypted_Bcast`, recall that each encrypted broadcast consists of an ordinary `MPI_Bcast` and an encryption/decryption per node. Hence the encryption overhead of the three encrypted MPI libraries, illustrated in Fig. 7, loosely mirrors their encryption-decryption throughput of Fig. 2.

- For example, for large messages (say 2MB), the encryption-decryption throughput of BoringSSL (1381 MB/s) is around 2.37 times that of Libsodium (583 MB/s). On the other hand, the encryption overhead in `Encrypted_Bcast` of BoringSSL (44.8%) is 2.03 times smaller than that of Libsodium (90.96%), approximating the ratio 2.37 above.
- As another example, for BoringSSL, the encryption-decryption throughput for 2MB messages is about the same as that for 16KB messages. Thus one would expect the encryption cost for 4MB messages in `Encrypted_Bcast` would be about $\frac{4\text{MB}}{16\text{KB}} = 256$ times that for 16KB messages. Indeed, for 4MB messages, BoringSSL spends about 4,298 μs on encryption/decryption, which is about 298 times its encryption/decryption time for 16KB messages (14.42 μs).

The trend of `Encrypted_Alltoall`, illustrated in Fig. 8, is similar.

- For example, for 16KB messages, the encryption-decryption throughput of BoringSSL (1332 MB/s) is about 2.35 times that of CryptoPP (568 MB/s). The encryption overhead of BoringSSL in `Encrypted_Alltoall` (17.19%) is 2.57 times smaller than that of CryptoPP (44.19%).
- As another example, for CryptoPP, the encryption-decryption throughput for 2MB messages (273 MB/s) is about a half of that for 16 KB messages (568 MB/s). Thus one would expect the encryption cost for 4MB messages in `Encrypted_Alltoall` would be about $\frac{4\text{MB}}{16\text{KB}} \cdot 2 = 512$ times that for 16KB messages. Indeed, for 4MB messages, CryptoPP spends about 1,331,103 μs on encryption/decryption, which is about 459 times its encryption/decryption time for 16KB messages (2900 μs).

Fig. 7: Encryption overhead (256-bit key), drawn in log scale, for `Encrypted_Bcast` on Ethernet.

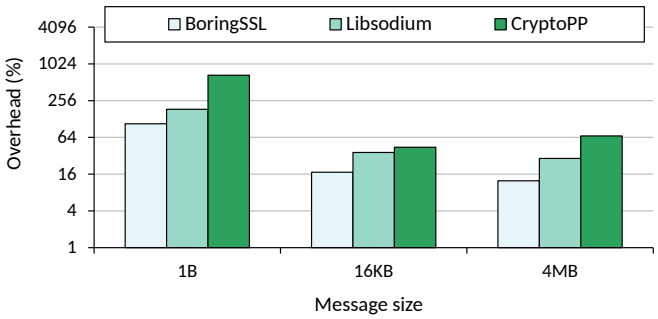

Fig. 8: Encryption overhead (256-bit key), drawn in log scale, for `Encrypted_Alltoall` on Ethernet.

NAS BENCHMARKS. To understand encryption overheads in a more realistic setting, we evaluated the encrypted MPI libraries under NAS parallel benchmarks. The results are shown in Table IV. Overall, BoringSSL's total running time is 99.81 seconds, whereas the baseline's running time is 88.52 seconds, and thus BoringSSL's overhead is 12.75%.[2] Likewise, Libsodium's and CryptoPP's overhead are 19.25% and 30.33% respectively. These results again support our thesis that encryption overheads may not be prohibitive for realistic scenarios where there are multiple concurrence communication flows.

B. Infiniband Results

ENCRYPTION-DECRYPTION. It turns out that the MVAPICH2-2.3 compiler, even with the same O2 flag, results in higher encryption-decryption performance than the gcc 4.8.5 compiler for some libraries. Fig. 9 shows the average encryption-decryption throughput of AES-GCM-256 code compiled by

[2]Conventionally, one would compute BoringSSL's overhead of each benchmark (BT, CG, FT, etc) and then report the average of them as BoringSSL's average overhead. However, as pointed out by several papers [20], [21], averaging over ratios is *meaningless*. Following the recommendation of those papers, here we instead derived BoringSSL's overhead from its total timing of all NAS benchmarks and that of the baseline.

	CG	FT	MG	LU	BT	SP	IS
Unencrypted	7.01	12.04	2.55	18.04	22.83	21.99	4.06
BoringSSL	8.55	12.81	3.01	19.05	27.40	24.46	4.52
Libsodium	9.62	13.67	3.09	19.48	28.70	26.30	4.71
CryptoPP	11.67	15.53	3.33	23.13	29.52	27.37	4.83

TABLE IV: Average running time (seconds) of NAS parallel benchmarks, Class C, 64-rank and 8-node, on Ethernet.

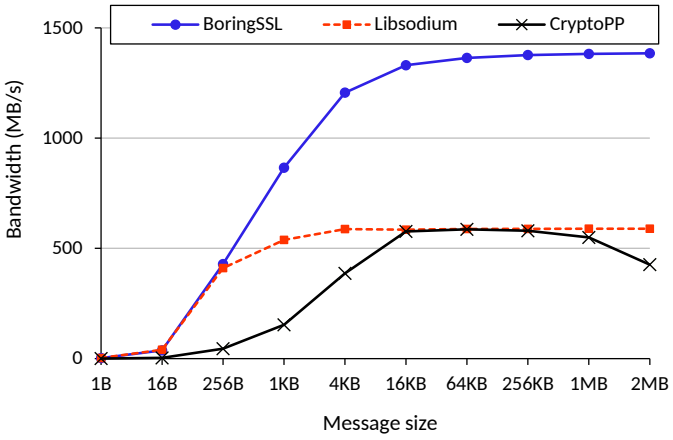

Fig. 9: Encryption-decryption throughput of AES-GCM-256, compiled under MVAPICH2-2.3.

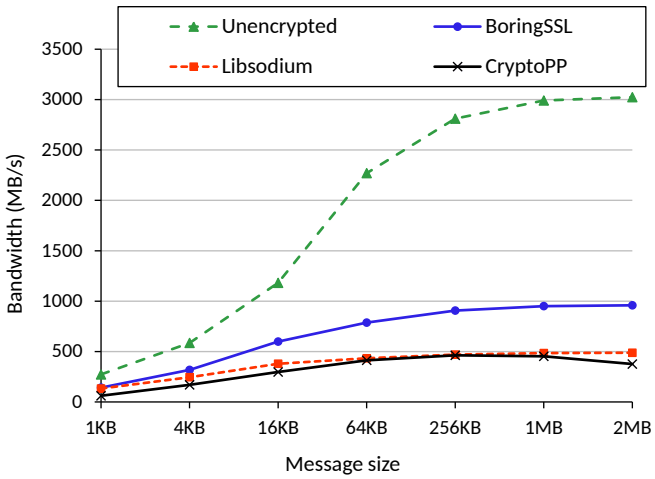

Fig. 10: Unidirectional ping-pong throughput with 256-bit encryption key on Infiniband, for medium and large messages.

the MVAPICH compiler. In particular, the performance of CryptoPP for message size greater than 64KB is dramatically improved. It seems natural to predict that while CryptoPP is still the last among the three encrypted MPI libraries, for large messages, its performance will be close to that of Libsodium.

PING-PONG. The ping-pong performance of the baseline and the encrypted MPI libraries is shown in Table V for small messages, and illustrated in Fig. 10 for medium and large messages.

Again, for large messages, the performance of the encrypted MPI libraries is much lower than that of the baseline, but the situation is much worse than the Ethernet setting. For example, for 2MB messages, even BoringSSL results in a 215.2% overhead. With InfiniBand, the baseline ping-pong throughput is significantly higher than that with Ethernet while the encryption-decryption throughput remains the same: the encryption-decryption throughput of AES-GCM-256 is much lower than the ping-pong throughput of the baseline. For example, for 2MB messages, the encryption-decryption throughput of AES-GCM-256 in BoringSSL (1384 MB/s) is just around 46% of the baseline ping-pong throughput (3023 MB/s), and thus estimatedly, BoringSSL's ping-pong time would be about $\frac{1+0.46}{0.46} \approx 3.17$ times slower than that of the baseline. This is consistent with the reported 215.2% overhead above.

	1B	16B	256B	1KB
Unencrypted	0.57	9.61	82.34	272.84
BoringSSL	0.22	4.02	45.51	142.23
Libsodium	0.27	4.86	50.66	133.06
CryptoPP	0.05	0.98	17.27	61.08

TABLE V: Average unidirectional ping-pong throughput (MB/s) for small messages, with 256-bit encryption key on Infiniband.

For small messages, the situation is somewhat better, but even BoringSSL would yield poor performance. For example, for 256-byte messages, BoringSSL has a 80.93% overhead.

OSU MULTIPLE-PAIR BANDWIDTH. The Multiple-Pair performance of the baseline and the encrypted MPI libraries, for 1B, 16KB, and 2MB messages, is shown in Figures 11, 12, and 13, respectively.

Like the Ethernet setting, for medium and large messages, although the encryption overhead is substantial when there is only one pair of communication, when the number of pairs increases, the throughput of encrypted MPI libraries is much closer to the baseline throughput. However, for medium message size (say 16KB), even when there are eight communication flows, BoringSSL only achieves 2561 MB/s,

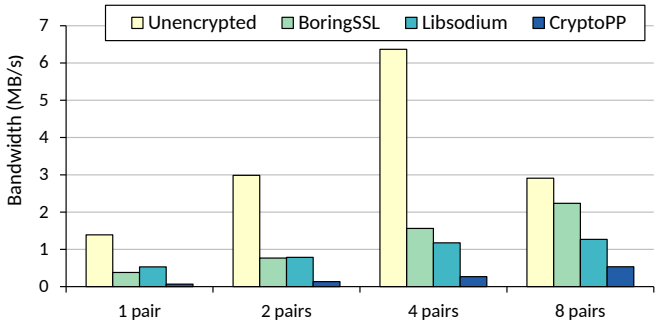

Fig. 11: OSU Multiple-Pair average throughput for 1B-messages on Infiniband.

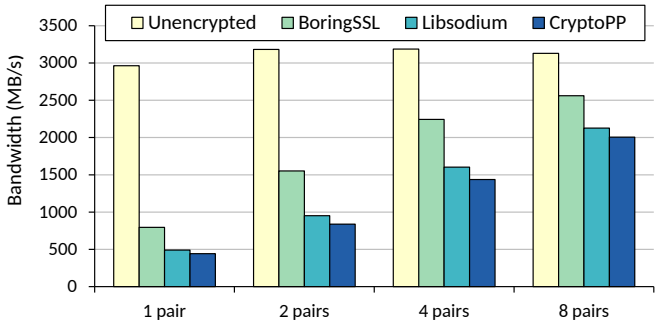

Fig. 12: OSU Multiple-Pair average throughput for 16KB-messages on Infiniband.

which is just 81.8% of the baseline throughput of 3128 MB/s. This gap is due to the speed difference between Ethernet and Infiniband.

For small messages, the trend is at first similar to that of the Ethernet setting, but when the number of pairs increases from 4 pairs to 8 pairs, the baseline throughput is throttled, probably due to network contention. This decrease also happens for medium and large messages, but not as conspicuously as the case of small messages. Due to the plummeting of the baseline throughput, for 8 pairs and one-byte messages, BoringSSL's overhead is just 29.91%, whereas for 4 pairs, its overhead is 308.33%.

COLLECTIVE COMMUNICATION. The average running time of `Encrypted_Bcast` and `Encrypted_Alltoall` for the 64-rank and 8-node setting, is shown in Tables VI and VII respectively. The trend, as illustrated in Fig. 14 for `Encrypted_Bcast` and Fig. 15 for `Encrypted_Alltoall`, is similar to that of the Ethernet setting, but the overhead is much worse, because Infiniband latency is lower.

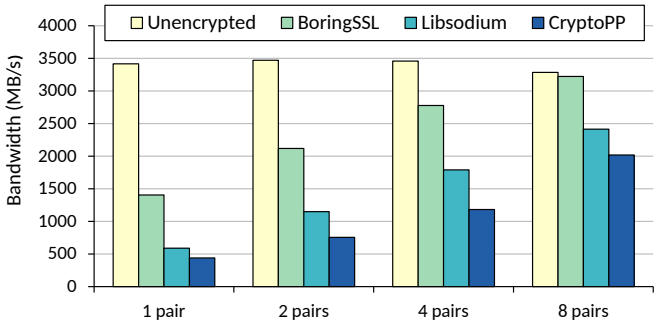

Fig. 13: OSU Multiple-Pair average throughput for 2MB-messages on Infiniband.

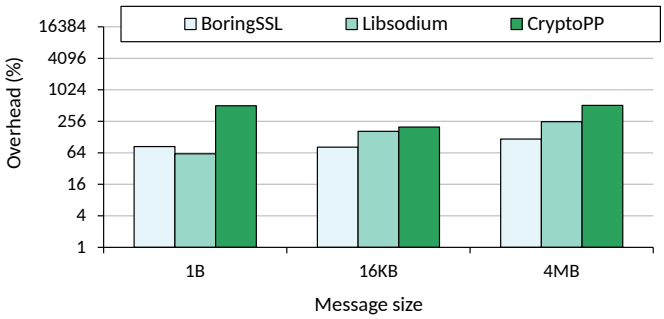

Fig. 14: Encryption overhead (256-bit key), drawn in log scale, of `Encrypted_Bcast` on Infiniband.

	1B	16KB	4MB
Unencrypted	4.14	28.58	3,780.27
BoringSSL	7.64	52.08	8,204.73
Libsodium	6.68	75.81	13,294.35
CryptoPP	25.25	85.43	23,344.63

TABLE VI: Average timing of `Encrypted_Bcast` (μs), with 256-bit encryption key on Infiniband.

	1B	16KB	4MB
Unencrypted	21.48	5,352.84	657,145.51
BoringSSL	435.70	6,789.17	1,013,896.50
Libsodium	736.29	7,977.41	1,305,389.60
CryptoPP	1,187.75	8,744.08	2,049,864.38

TABLE VII: Average timing of `Encrypted_Alltoall` (μs), with 256-bit encryption key on Infiniband.

NAS BENCHMARKS. The results of NAS benchmarks for Infiniband are shown in Table VIII. Here the overheads of BoringSSL, Libsodium, and CryptoPP are 17.93%, 24.27% and 29.41% respectively. CryptoPP's overhead in Infiniband is

	CG	FT	MG	LU	BT	SP	IS
Unencrypted	6.55	10.00	3.59	18.36	24.56	24.20	3.04
BoringSSL	8.36	10.77	4.20	19.73	33.35	26.87	3.20
Libsodium	9.87	11.52	4.28	20.04	34.62	28.55	3.33
CryptoPP	10.47	11.89	4.41	22.82	34.96	28.97	3.35

TABLE VIII: Average running time (seconds) of NAS parallel benchmarks, Class C, 64-rank and 8-node, on Infiniband.

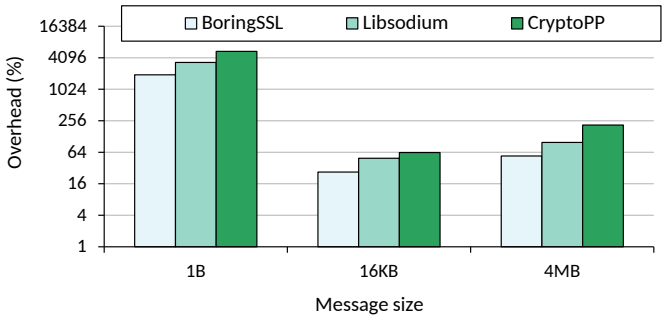

Fig. 15: Encryption overhead (256-bit key), drawn in log scale, of `Encrypted_Alltoall` on Infiniband.

slightly less than that in Ethernet, because the compiler in the former setting uses more aggressive optimizations than its Ethernet counterpart, which drastically improves the performance of CryptoPP, as shown in Figures 2 and 9. These results again reiterate our thesis that even in very fast networks, encryption overheads may not be prohibitive for practical scenarios.

C. Discussion

As shown in the experiments, in several settings, there is a large gap between the performance of the encrypted MPI libraries and the baseline. This happens because as shown in Figures 2 and 9, the *single-thread* encryption speed is not fast enough to support the high-speed network in data centers, which have reached 100Gbps. This problem will likely be worse in the future since it is expected that the network bandwidth will continue increasing while the CPU single-thread performance will not have significant improvement. To fully utilize the network links whose throughput is significantly higher than the single thread encryption-decryption throughput, one will almost have no choice but to parallelize encryption using multiple threads, or accelerate it via GPU.

VI. Conclusion

We considered adding encryption to MPI communications for providing privacy and integrity. Four widely used cryptographic libraries, OpenSSL, BoringSSL, CryptoPP, and Libsodium, were studied in this paper. We found that the encryption overhead differs drastically across libraries, and that BoringSSL (and OpenSSL) achieves the best performance in most settings. Moreover, when individual communication is considered, encryption overhead can be quite large. However, in practical scenarios when multiple communication flows are carried out concurrently, the overhead is not significant. In particular, our evaluation with the NAS parallel benchmarks shows that using the best cryptographic library BoringSSL, our implementation on average only introduces 12.75% overhead on Ethernet and 17.93% overhead on Infiniband.

Acknowledgment

We thank Sriram Keelveedhi for helpful discussions, the anonymous reviewers of IEEE Cluster 2019 for insightful feedback, and Prof. Weikuan Yu at Florida State University for providing the computing infrastructure for software development and performance measurement.

References

[1] X. Ruan, Q. Yang, M. I. Alghamdi, S. Yin, and X. Qin. ES-MPICH2: A Message Passing Interface with enhanced security. IEEE Trans. Dependable Secur. Comput., 9(3):361374, May 2012.

[2] J. Katz and Y. Lindell. Introduction to modern cryptography. CRC press, 2014.

[3] OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org, 2018.

[4] BoringSSL. https://boringssl.googlesource.com/boringssl, 2018.

[5] The Sodium cryptography library (Libsodium). https://libsodium.gitbook.io/doc, 2018.

[6] W. Dai. CryptoPP. https://www.cryptopp.com, 2018.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Int. J. High Perform. Comput. Appl., 5(3):6373, Sept. 1991.

[8] M. J. Dworkin. NIST SP 800-38D. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. 2007.

[9] B. Balamurugan, P. V. Krishna, G. V. Rajya Lakshmi, and N. S.Kumar. Cloud cluster communication for critical applications accessing C-MPICH. In 2014 International Conference on Embedded Systems (ICES 2014), pages 145150, July 2014.

[10] M. A. Maffina and R. S. RamPriya. An improved and efficient message passing interface for secure communication on distributed clusters. In 2013 International Conference on Recent Trends in Information Technology (ICRTIT 2013), pages 329334, July 2013.

[11] V. Rekhate, A. Tale, N. Sambhus, and A. Joshi. Secure and efficient message passing in distributed systems using one-time pad. In 2016 International Conference on Computing, Analytics and Security Trends (CAST 2016), pages 393397, Dec 2016.

[12] S. Shivaramakrishnan and S. D. Babar. Rolling curve ECC for centralized key management system used in ECC-MPICH2. In 2014 IEEE Global Conference on Wireless Computing Networking (GCWCN 2014), pages 169173, Dec 2014.

[13] J. Mason, K. Watkins, J. Eisner, and A. Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In Proceedings of the 13th ACM conference on Computer and communications security (CCS 2006), pages 235244. ACM, 2006.

[14] J. H. An and M. Bellare. Does encryption with redundancy provide authenticity? In International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2001), pages 512528. Springer, 2001.

[15] D. A. McGrew and J. Viega. The security and performance of the Galois/Counter Mode (GCM) of operation. In International Conference on Cryptology in India (INDOCRYPT 2004), pages 343355. Springer, 2004.
[16] M. J. Dworkin. NIST SP 800-38A. Recommendation for block cipher modes of operation: Methods and techniques. 2001.
[17] T. Krovetz and P. Rogaway. The software performance of authenticatedencryption modes. In International Workshop on Fast Software Encryption (FSE 2011), pages 306327. Springer, 2011.
[18] The Heartbleed Bug. http://heartbleed.com/.
[19] OSU Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.
[20] P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. Communications of the ACM, 29(3):218221, 1986.
[21] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In Proceedings of the international conference for high performance computing, networking, storage and analysis (SC 2015), page 73. ACM, 2015.