HARDWARE INSTRUCTION BASED CRC32C, A BETTER ALTERNATIVE TO THE TCP ONE'S COMPLEMENT CHECKSUM

A Thesis

by

SAURAV KUMAR SAHU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2019

Major Subject: Electrical Engineering

ABSTRACT

End-to-end data integrity is of utmost importance when sending data through a communication network, and a common way to ensure this is by appending a few bits for error detection (e.g., a checksum or cyclic redundancy check) to the data sent. Data can be corrupted at the sending or receiving hosts, in one of the intermediate systems (e.g., routers and switches), in the network interface card, or on the transmission link. The Internet's Transmission Control Protocol (TCP) uses a 16-bit one's complement checksum for end-to-end error detection of each TCP segment [1]. The TCP protocol specification dates back to the 1970s, and better error detection alternatives exist (e.g., Fletcher checksum, Adler checksum, Cyclic Redundancy Check (CRC)) that provide higher error detection efficiency; nevertheless, the one's complement checksum is still in use today as part of the TCP standard. The TCP checksum has low computational complexity when compared to software implementations of the other algorithms. Some of the original reasons for selecting the 16-bit one's complement checksum are its simple calculation, and the property that its computation on big- and little-endian machines result in the same checksum but byte-swapped. This latter characteristic is not true for a two's complement checksum. A negative characteristic of one's and two's complement checksums is that changing the order of the data does not affect the checksum. In [2], the authors collected two years of data and concluded after analysis that the TCP checksum "will fail to detect errors for roughly one in 16 million to 10 billion packets." While some of the sources responsible for TCP checksum errors have decreased in the nearly 20 years since this study was published (e.g., the ACK-of-FIN TCP software bug), it is not clear what we would find if the study were repeated. It would also be difficult to repeat this study today because of privacy concerns. The advent of hardware CRC32C instructions on Intel x86 and ARM CPUs offers the promise of significantly improved error detection (probability of undetected errors proportional to 2^{-32} versus 2^{-16}) at a comparable CPU time to the one's complement checksum.

The goal of this research is to compare the execution time of the following error detection algorithms: CRC32C (using generator polynomial 0x1EDC6F41), Adler checksum, Fletcher checksum, and one's complement checksum using both software and special hardware instructions. For CRC32C, the software implementations tested were bit-wise, nibble-wise, byte-wise, slicing-by-4 and slicing-by-8 algorithms. Intel's CRC32 and PCLMULQDQ instructions and ARM's CRC32C instruction were also used as part of testing hardware instruction implementations. A comparative study of all these algorithms on Intel Core i3-2330M shows that the CRC32C hardware instruction implementation is approximately 38% faster than the 16-bit TCP one's complement checksum at 1500 bytes, and the 16-bit TCP one's complement checksum is roughly 11% faster than the hardware instruction based CRC32C at 64 bytes. On the ARM Cortex-A53, the hardware CRC32C algorithm is approximately 20% faster than the 16-bit TCP one's complement checksum at 64 bytes, and the 16-bit TCP one's complement checksum is roughly 13% faster than the hardware instruction based CRC32C at 1500 bytes. Because the hardware CRC32C instruction is commonly available on most Intel processors and a growing number of ARM processors these days, we argue that it is time to reconsider adding a TCP Option to use hardware CRC32C.

The primary impediments to replacing the TCP one's complement checksum with CRC32C are Network Address Translation (NAT) and TCP checksum offload. NAT requires the recalculation of the TCP checksum in the NAT device because the IPv4 address, and possibly the TCP port number change, when packets move through a NAT device. These NAT devices are able to compute the new checksum incrementally due to the properties of the one's complement checksum. The eventual transition to IPv6 will hopefully eliminate the need for NAT. Most Ethernet Network Interface Cards (NIC) support TCP checksum offload, where the TCP checksum is computed in the NIC rather than on the host CPU. There is a risk of undetected errors with this approach since the error detection is no longer end-to-end; nevertheless, it is the default configuration in many operating systems including Windows 10 [3] and MacOS. CRC32C has been implemented in some NICs to support the iSCSI protocol, so it is possible that TCP CRC32C offload could be supported in the future. In the near term, our proposal is to include a TCP Option for CRC32C in addition to the one's complement checksum for higher reliability.

DEDICATION

To my parents and friends for their love, faith, encouragement, and emotional support without which this thesis would have never completed.

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to my thesis advisor Dr. Pierce E. Cantrell for his unwavering support throughout the duration of this research. His patience is really commendable. He has always encouraged me to experiment with new things during the research. He has also provided me with most of the resources that helped me in conducting my research smoothly. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

Once again, I would like to thank Dr. Cantrell for giving me the opportunity of being a Teaching Assistant for ECEN 248 and ECEN 602. I really enjoyed interacting with several students during the course of teaching, and it certainly enriched my grad school experience. Next, I would like to offer my sincere thanks to Dr. Srinivas Shakkottai, Dr. Krishna R. Narayanan, and Dr. Andreas Klappenecker for taking their valuable time to conduct my thesis defense and providing me with pointers for further improvement. I would like to acknowledge the assistance that I had received from Dr. Paul Gratz regarding the installation of 64-bit Gentoo Linux OS on Raspberry Pi Model 3B. I very much appreciate Katie Bryan and the University Writing Center for reviewing my thesis document.

It is ultimately the friends who make the graduate school experience most enjoyable. I would like to extend my heartfelt thanks to Abhijeet Sahu who has always inspired me throughout the grad school journey. Thank you once again Abhijeet for standing by my side no matter what. Special thanks to Tushar, Mahesh, Sai, Soumya, and Vijith for being awesome roommates and for helping me with everything. Many thanks to Dharanidhar Bhai and Inderdeep for motivating me to do exercise and pranayam. I am really thankful for all the resources and opportunities that I got in Texas A&M University which enhanced my knowledge and made me a better person. Thanks should also go to NTPC Ltd. for giving me the opportunity to pursue a master's degree. Finally, I would like to thank the Almighty and my parents for bearing with me.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. Pierce E. Cantrell [advisor], Dr. Srinivas Shakkottai and Dr. Krishna R. Narayanan of the Department of Electrical and Computer Engineering and Dr. Andreas Klappenecker of the Department of Computer Science and Engineering.

Most of the research work for the thesis was completed by me under the supervision of Dr. Pierce E. Cantrell.

Funding Sources

Graduate study was partly supported by Teaching Assistantship and Research Assistantship from Texas A&M University.

NOMENCLATURE

API	Application Programming Interface
BER	Bit Error Rate
BSC	Binary Symmetric Channel
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
FCS	Frame Check Sequence
GCC	GNU Compiler Collection
GF(2)	Galois Field of two elements (0 and 1)
HD	Hamming Distance
HW	Hamming Weight
IP	Internet Protocol
ISCSI	Internet Small Computer System Interface
KiB	Kibi Byte
LSb	Least Significant bit
LSB	Least Significant Byte
LUT	Look up Table
MSb	Most Significant bit
MSB	Most Significant Byte
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
NASM	The Netwide Assembler
NAT	Network Address Translation

NIC	Network Interface Controller
OS	Operating System
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PRNG	Pseudo Random Number Generators
PSTN	Public Switched Telephone Network
P_{ud}	Probability of undetected error
RFC	Request for Comments
SoC	System-on-Chip
SCTP	Stream Control Transmission Protocol
SSE	Streaming SIMD Extensions
SIMD	Single instruction multiple data
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XOR	Exclusive-OR

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

1.1 Motivation

The Transmission Control Protocol (TCP) is an end-to-end protocol that provides communication between processes running on end nodes. TCP also offers a connection-oriented and reliable byte-stream service. It is because of this reliable and in-order delivery of a stream of bytes that the application running on top of TCP does not have to worry about out-of-order or missing data packets. Moreover, the error detection algorithms operating in the different layers of the OSI (Open Systems Interconnection) model provide the integrity of the header and the data. The 16-bit one's complement checksum, which is a relatively weak error detection scheme when compared to the 32-bit Cyclic Redundancy Check (CRC), is currently used to detect end-to-end errors at the receiving end of a TCP connection. If there are errors in a TCP segment, the hope is that the TCP checksum will detect the error. Many applications simply rely on the existing TCP checksum to provide data integrity and do not employ any additional application-level error detection.

Data corruption to a TCP segment can occur in the source node due to memory errors, software errors in the TCP/IP stack, errors in the network interface controller (NIC), errors on the wire while data is in transit, hardware errors in Ethernet switch, errors in intermediate router memory or hardware, errors in the destination node NIC, or errors in memory or the TCP/IP stack running on the end node. The Ethernet Frame Check Sequence (FCS) is a 32-bit CRC that detects errors in frames received over a link. While the link-level CRC provides additional protection against layer-2 errors, it is no substitute for end-to-end error detection. The IPv4 header checksum is responsible for the protection of the header of IPv4 packets against errors. Following the end-to-end principle of error detection, the two end nodes across the communication network can achieve a reliable data transfer by sending a checksum calculated over the TCP header, pseudo header and the payload data and then recalculating the same at the destination end to check for correctness. There may be undetected errors, but many errors will be detected.

Stone et al. [2] have strongly urged that critical applications append application-level checksum to data in addition to the TCP checksum. The rationale behind their proposition is that the TCP checksum does not offer sufficient protection against an undetected error. While the data of [2] on TCP segments received with checksum errors is nearly 20 years old, there are a handful of recent reports of undetected TCP checksum errors [4]. With the adoption of Jumbo Frames carrying 9000 bytes of payload in the data center environment, the number of undetected errors is bound to increase. TCP being central to most of the applications communicating over the Internet, implementing a stronger error detection scheme in TCP itself is a preferred option.

Although many checksum techniques exist in the literature and a wide variety are in use in a multitude of protocols, less information is available about their relative execution speeds in terms of the number of CPU cycles per byte. Historically, for data integrity checking purpose for a communication network, the CRC has not been used above the data link layer because it is computationally expensive in software. Two notable exceptions above the link layer that use a 32-bit CRC are the Internet Small Computer Systems Interface (iSCSI) [5] and the Stream Control Transmission Protocol (SCTP) [6]. As new network protocols are constantly emerging, it is worthwhile to know which checksum approach works best in a given processor configuration. Knowing both the software and hardware instruction implementation schemes is therefore desirable to make a better trade-off.

Among all the Intel processors, the ones based on the Nehalem microarchitecture were the first to have a CRC32 instruction (uses Castagnoli polynomial 0x1EDC6F41) as part of the SSE4.2 instruction set [7]. The Nehalem Intel processors were released in 2008 [8], and subsequent Intel processors support SSE4.2 including CRC32C. Modern ARM processors also have hardware CRC32C instruction (uses Castagnoli polynomial 0x1EDC6F41) available in architecture ARMv8.1-A (December 2014) and later. The CRC32C instruction is optionally supported in ARMv8-A [9] as well. The Apple A12 Bionic, a system on a chip (SoC) present in iPhone XS, XS Max, and XR has 64-bit ARMv8.3-A six-core CPU, which presumably support a hardware CRC instruction. The Raspberry Pi 3 and modern Android devices like the Nexus 5X and Google

Pixel featuring ARMv8-A CPU have a CRC32 hardware instruction. The IBM POWER8 provides hardware accelerated support for CRC32 using the vpmsum (vector polynomial multiply sum) instruction. We can make use of the hardware CRC32 instruction to compute the CRC of the header and payload of a packet in emerging protocols, and possibly retrofit standard protocols like TCP with CRC32 for significantly improved error detection performance.

The primary impediments to replacing the TCP one's complement checksum with CRC32C are Network Address Translation (NAT) and TCP checksum offload. NAT requires the recalculation of the TCP checksum in the NAT device because the IPv4 address, and possibly the TCP port number change, when packets move through a NAT device. These NAT devices are able to compute the new checksum incrementally due to the properties of the one's complement checksum. The eventual transition to IPv6 will hopefully eliminate the need for NAT. Most Ethernet Network Interface Cards (NIC) support TCP checksum offload, where the TCP checksum is computed in the NIC rather than on the host CPU. There is a risk of undetected errors with this approach since the error detection is no longer end-to-end; nevertheless, it is the default configuration in many operating systems including Windows 10 [3] and MacOS. CRC32C has been implemented in some NICs to support the iSCSI protocol, so it is possible that TCP CRC32C offload could be supported in the future. In the near term, our proposal is to include a TCP Option for CRC32C in addition to the one's complement checksum for higher reliability.

1.2 Overview and Organization

This thesis explores widely utilized checksums in the literature including the following: one's complement Internet checksum, Fletcher checksum, Adler checksum, and CRC32C (Castagnoli Polynomial 0x1EDC6F41 [10]). The CRC is not technically a checksum as it uses base two polynomial division to calculate the remainder. However, the term checksum is generically used in the literature. For the one's complement checksum and CRC32C, we have examined both software and hardware instruction implementations (i.e., add-with-carry instruction for one's complement addition and the CRC32C instruction for CRC). We have done a comparative performance evaluation of these algorithms to determine how many CPU cycles per byte each algorithm takes.

Our results show that a CRC32C implementation using a hardware instruction has the same order of CPU cycles per byte as does the 16-bit one's complement checksum. This opens up the possibility of using CRC32C instead of the 16-bit one's complement checksum for much stronger error detection in TCP when the corresponding hardware instruction is available. We propose using a TCP Option to negotiate the use of the hardware CRC. The enhanced error detection with improved Hamming Distance (HD) and significantly lower probability of undetected error is well worth the price of the extra 16 bits per TCP segment.

In Chapter 2, we introduce error detection describing some of the characteristics of the error detection algorithms, and we review the literature. In Chapter 3, we introduce the theory behind the CRC, and discuss why we recommend using CRC32C in TCP. We also state some of the useful properties of the CRC.

Chapter 4 of this thesis explains the working details of all the algorithms that have been considered in this research (e.g., one's complement checksum, Adler-32 checksum, Fletcher-32 checksum, and many variants of the CRC32C algorithms). We discuss the details of the benchmark routine that we have used in our research in Chapter 5. We also discuss how we carried our tests across different platforms. We present the experimental results in the Chapter 6. In Chapter 7, we propose the use of a TCP Option in order to incorporate the CRC32C in TCP, and we finally present our conclusions in the Chapter 8.

2. BACKGROUND AND RELATED WORK

2.1 Error Detection

Everything we transmit over the Internet is a string of zeroes and ones. A bit is said to be in error when either it is flipped from one to zero or vice-versa. Bit errors show up occasionally due to electrical interference or noise, switch/router memory errors, faulty Network Interface Controller (NIC), etc. These errors need to be detected using some mechanism; otherwise, the data received at the end node might be accepted in error. A checksum is used in TCP to preserve data integrity. The checksum is calculated by summing up all the bytes or words taken from the stream of bytes, and we send it as additional bits along with the data bits. The destination node recomputes the checksum, and if it matches with the one that is received, then with some probability we would infer that the received stream of bytes is error free. However, it is also possible that bit errors are introduced in both the data and/or the checksum in such a way that the modified stream of bytes sums up to a modified checksum leading to an undetected error. There is a trade-off among the checksum algorithm's computational complexity, number of bits used for the checksum field, and the probability of undetected errors (P_{ud}) [11].

CRC32 is used in Ethernet for error detection in the received frame. A 32-bit generator polynomial is used to generate the CRC, which is appended as a trailer after the end of the message. Mathematically, a CRC is generated by treating a stream of bits as a polynomial over GF(2) (i.e. with the polynomial coefficients being 0 or 1) and dividing that polynomial by a generator polynomial. The quotient of that division is discarded and the remainder is the desired Frame Check Sequence (FCS). Validation of the FCS [12] is done by comparing the FCS calculated at the receiver end with the FCS that is received along with the rest of the data. If they do not match, then an error has occurred, and the frame is treated as invalid. Based on experimental results, all of the CRCs appear to be data independent; whereas, addition-based checksums, like the one's complement and Fletcher checksums, have shown data dependencies [11]. The CRCs are considered data independent, because the error detection performance is not affected by the data-word values. However, as the probability of undetected error is affected by the distribution of data-word values for the addition-based checksums, they are considered data dependent [11].

2.2 Error detection code effectiveness measures

2.2.1 Hamming Distance

The Hamming Distance (HD) of an error detection algorithm is defined as the minimum number of bit errors required to convert one valid codeword into another valid codeword. A code is said to detect n number of bit errors if the minimum HD between any two valid codewords is at least $n + 1$. For example, an error detection code with HD=3 would be able to detect all possible one and two-bit errors but would miss at least one three-bit error out of the entire space of three-bit errors. Keeping the checksum size fixed, the Hamming Distance of the error detection code in general depends on the length of the data. If we increase the length of the data, then the HD would decrease.

2.2.2 Hamming Weight

For an error detection code, the Hamming Weight (HW) for a certain number of bits is equal to the number of undetectable errors with that number of bits in error [11]. Assume that the length of a sample codeword is 100 bits. The number of three-bit errors that could possibly occur is $\binom{100}{3}$ = $\frac{100!}{3!(100-3)!}$ = 161,700. Let us further assume that out of 161,700 different possible combinations of three-bit errors, the error detection code fails to detect 5,000 of such three-bit errors. Therefore, HW_3 is equal to 5,000. If the HW is zero for some number of bits, then all the errors would be detected for that number of bits.

2.2.3 Probability of undetected error

The probability of undetected error (P_{ud}) for an error detection algorithm is the summation of probabilities of each number of undetected bit errors. For example, if the codeword length is 10 bits and the minimum HD between any two valid codewords is 6, then

$$P_{\text{ud}} = \frac{HW_6}{\binom{10}{6}} + \frac{HW_7}{\binom{10}{7}} + \frac{HW_8}{\binom{10}{8}} + \frac{HW_9}{\binom{10}{9}} + \frac{HW_{10}}{\binom{10}{10}}$$

This is true because all the 1-bit, 2-bit, 3-bit, 4-bit and 5-bit errors would be detected for our example error detection code.

2.3 Literature Review

Stone et al. [2] in 2000 noted that "1 datagram in 7,500 passed the link-level CRC but failed the TCP or UDP checksum." They have conjectured that the data was corrupted somewhere in the intermediate systems (e.g., router) or in the source and destination nodes but not on the transmission link as the link-level CRC would catch the overwhelming number of these errors. After analyzing many Internet data traces, they have characterized different types of errors and their sources. "Do not trust the hardware" was one of the highlighted points of their study. Finally, they have recommended that the crucial applications should use an application-level checksum in addition to the TCP checksum to better protect the valuable data. It would be interesting to repeat this study to see if these error rates have decreased in the last 20 years, but these days the privacy concerns would be difficult to overcome. Certainly error detection in computer system has improved in the last 20 years. For example, the PCI Express (PCIe) serial bus interconnecting the CPU and the NIC includes a 32-bit CRC for error detection, and high performance Internet backbone routers now support ECC memory. Nevertheless, there are still many places for errors outside the link layer to occur.

Jones mentions an outage of Amazon's S3 storage system in 2008 because of a single-bit corruption in some of the messages leading to server-to-server communication problems [4]. Manav mentions Open Shortest Path First (OSPF) corrupted packets that successfully sneaked past the CRC and IP checksum without getting caught and were delivered to the OSPF stack causing problems [13]. Jones also describes a bug in the veth kernel module that caused the packets corrupted by a faulty Ethernet switch to be ignored and silently delivered to the applications in Twitter without the TCP checksum being verified and rejected by software.

Saltzer, Reed, and Clark in their classic 1981 paper “End-to-end arguments in system design” [14] make the case for “end-to-end check and retry.” They provide “a too-real example” to illustrate their case.

> An interesting example of the pitfalls that one can encounter turned up recently at M.I.T.: One network system involving several local networks connected by gateways used a packet checksum on each hop from one gateway to the next, on the assumption that the primary threat to correct communication was corruption of bits during transmission. Application programmers, aware of this checksum, assumed that the network was providing reliable transmission, without realizing that the transmitted data was unprotected while stored in each gateway. One gateway computer developed a transient error in which while copying data from an input to an output buffer a byte pair was interchanged, with a frequency of about one such interchange in every million bytes passed. Over a period of time many of the source files of an operating system were repeatedly transferred through the defective gateway. Some of these source files were corrupted by byte exchanges, and their owners were forced to the ultimate end-to-end error check: manual comparison with and correction from old listings [14].

Another real-world incident took place in May and June of 1980 when the failure of a single chip on a Network Interface Card (NIC) resulted in false missile launch warnings being sent from the North American Aerospace Defense Command (NORAD) in Colorado Springs to the National Military Command Center (NMCC) in the Pentagon and the Strategic Air Command (SAC) headquarter in Omaha [15] [16]. The NORAD incident, which did not result in WWIII thanks to having humans in the decision loop, was due to “checksum offload.” While the original message was generated correctly on a mainframe with Error Correcting Code (ECC) memory, the message checksum was calculated in the communication interface board after the data had been corrupted intermittently by the bad chip. Consequently, the corrupted message had a good checksum.

We seem to have forgotten some of the lessons from the past on the necessity of end-to-end protection of data. In particular, TCP checksum offload and Network Address Translation (NAT) both violate the end-to-end principle. NAT requires the recalculation of the TCP checksum in the NAT device because the IPv4 address, and possibly the TCP port number change, when packets move through a NAT device. These NAT devices are able to compute the new checksum incrementally due to the properties of the one's complement checksum. While this is clearly a violation of the end-to-end principle, NAT will be a fact-of-life until IPv6 is universally adopted. The eventual transition to IPv6 will hopefully eliminate the need for NAT. The TCP checksum offload is supported by most Ethernet NIC, and is therefore enabled by most operating systems where the TCP checksum is computed in the NIC rather than on the host CPU. There is a risk of undetected errors with this approach since the error detection is no longer end-to-end; nevertheless, it is the default configuration in many operating systems including Windows 10 [3]. Foong et al. [17] in 2003 estimated that the TCP checksum offload saves about 10% CPU utilization when compared to software checksum and copy. In our informal tests, TCP checksum offload in Windows 10 with Gigabit Ethernet, when using the speedtest.net application connected to a campus server, resulted in approximately 10% increase in throughput.

Request for Comments (RFC) 1071 [18] presents methods to efficiently compute the 16-bit one's complement checksum that is used in Internet Protocol version 4 (IPv4), TCP and the User Datagram Protocol (UDP) for detecting errors. RFC 1071 also discusses how various checksum properties can be exploited to speed up the checksum calculation. Several numerical examples and CPU specific algorithms are also provided. The Motorola 68020 chip was reported to take approximately 2.617 cycles/byte for summing random data taken 32 bits at a time. The appendix in RFC 1071 characterizes the design features of the TCP checksum function. Several useful properties of one's complement addition are commutativity, associativity, existence of inverse and identity element, and incremental modification.

RFC 1146 [19] describes the 8-bit and 16-bit Fletcher checksum algorithms and describes a TCP Option mechanism to negotiate the use of these algorithms as an alternative to the 16-bit one's complement checksum for error detection. A TCP implementation not recognizing this

TCP Option should ignore it silently. However, as this TCP extension did not see widespread deployment, it was moved to historic status per RFC 6247 [20].

The Adler-32 checksum algorithm is discussed in detail in RFC 2960 [21]. Initially, this checksum algorithm was used in the Stream Control Transmission Protocol (SCTP), which was designed to carry Public Switched Telephone Network (PSTN) signaling messages over IP networks. However, later it was discovered that the Adler-32 checksum provides weak error detection for small packets (for example less than 128 bytes), and it was later replaced with the CRC32C [6]. While the Fletcher checksum performs the addition of a stream of bytes modulo 255 for 8-bit data word and modulo 65535 for 16-bit data word, the Adler checksum does addition modulo 65521 (the largest prime number below 65535) for 16-bit block sized data to avoid a possible large class of 16-bit errors that could go undetected [22].

RFC 3385 [5] attempts to find a stronger error detection algorithm for use in the iSCSI protocol. It compares CRCs with Fletcher, Adler and weighted checksum algorithms with probability of undetected error (P_{ud}) being the main criteria. The authors have separately analyzed the behaviour of the checksums in the presence of random independent single bit error and burst errors. They have cited a paper from 1998 by Stone et al. [23] which shows that the TCP checksum performs poorly on non-uniform real data. The authors of RFC 3385 have found that "for independent bit errors, P_{ud} of CRC is approximately 12,000 times better than the Fletcher checksum and 22,000 times better than the Adler checksum." They ultimately recommended the use of CRC32C as the error detection mechanism in the iSCSI since the CRC is data independent and offers a much lower probability of undetected error.

Koopman [24] found a new set of 32-bit CRC generator polynomials that provides a better Hamming Distance (HD) than the IEEE 802.3 32-bit CRC polynomial (0x82608EDB in the reversed reciprocal notation) for Ethernet MTU-sized frames. He has cleverly exploited some less computationally expensive filtering algorithms to conduct an exhaustive search for 32-bit CRCs with better error detection properties. He has studied the 32-bit CRC polynomial 0xBA0DC66B that achieves HD=6 up to a data word length of 16,360 bits (greater than the 12,000 bit typical Ethernet MTU sized frame) and HD=4 up to 114,663 bits (greater than the 72,000 bit Jumbo frame). The IEEE 802.3 CRC32 polynomial (0x04C11DB7) achieves HD=4 up to 91,607 bits and HD=6 up to 268 bits; whereas, the CRC32C polynomial (0x1EDC6F41) achieves HD=4 up to 131,072 bits and HD=6 up to 5,243 bits [24].

Maxino et al. [11] studied the effectiveness of the following error detection algorithms: Exclusive OR (XOR), Two's complement addition, One's complement addition, Adler checksum, Fletcher checksum and CRCs. Their area of interest is embedded control networks and the following metrics they used for comparing different checksum algorithms: the probability of undetected random independent bit errors in a binary symmetric channel (BSC), Hamming Distance and the probability of undetected burst errors. With the data dependent checksums (e.g., one's complement and two's complement addition, Fletcher and Adler checksum), the authors have shown that the maximum proportion of undetected errors occurs for random data that has an evenly distributed zeroes and ones in each bit position in the data block. They have emphasized that a good CRC has much better HD as compared to that of Adler and Fletcher checksum for a given length of data and for the equally sized checksums. Moreover, they have found that an optimal CRC with a fewer number of bits outperforms the Fletcher checksum (e.g, CRC-12 performs better than Fletcher-16 for every possible codeword length all the time). They have also noted that the CRC is more beneficial to use as compared to the other checksums especially when the length of the codeword is small. Finally, they have concluded that a good CRC polynomial should always be used whenever computational resources are available.

Daugherty [25] evaluated the risk associated with disabling the iSCSI digest which is essentially a 32-bit CRC. The iSCSI protocol transports the SCSI commands over the TCP/IP network connecting the data storage centers, and it has a provision for optional header and data digests. The author has expressed concern for the integrity of data in the absence of the iSCSI digest, as in such a case the end-to-end error detection is looked after by substantially weaker TCP 16-bit one's complement checksum. Moreover, the data corruption that takes place during the Protocol Data Unit (PDU) handover from TCP to iSCSI would go undetected if the iSCSI digest is not there.

They have mentioned that with iSCSI digests being enabled, the data transfer rate could go down by 5-20% and it is the job of network admin to decide on a case-by-case basis if the data transfer performance or the data itself is truly valuable.

Kounavis et al. [26] developed CRC slicing-by-4 and slicing-by-8 algorithms, which are software based table driven CRC implementations based on the Sarwate algorithm [27]. These two algorithms use precalculated look up tables of size 4 KiB (KibiByte) and 8 KiB respectively. The slicing-by-4 algorithm is able to read four bytes of data at a time; whereas, slicing-by-8 can read eight bytes of data at once. The CRC calculated over the first four or eight bytes is stored in an accumulator. This accumulator value is used as the initial CRC value to calculate the CRC for the next chunk of four or eight bytes and so on. They have reported that slicing-by-4 and slicing-by-8 algorithms are respectively two times and three times faster approximately than the Sarwate algorithm that calculates CRC cumulatively over the bytes taking only one byte of data at a time using a similar table look up approach.

Gueron [28] has used Intel's CRC32 instruction to transform the traditional latency-bound CRC32C computations into throughput oriented ones so as to maximize the utilization of the hardware in a pipelined fashion, achieving almost a three times increase in the computational performance. He has cleverly used the fact that CRC32 instruction in the Intel instruction set has latency of three clock cycles and throughput of just one clock cycle. Therefore, he carefully divides the data buffer into three parts and calculates the CRC32C of these parts one by one in a round-robin fashion avoiding the waiting time. Finally, he merges the results obtained from these three parts with some shift and XOR (Exclusive OR) operations to obtain the final resultant CRC32C over the entire buffer.

3. CRC32C

3.1 CRC Math

Cyclic Redundancy Check (CRC) is an error detection algorithm and the objective of the CRC is to maximize the error detection probability using a fairly small number of redundant bits. For example, Ethernet uses a 32-bit CRC polynomial 0x82608EDB to provide data integrity for a frame extending well beyond 1500 bytes.

The message that we want to exchange between the sender and receiver can be represented as a polynomial [29]. Let us assume that the message has $(n+1)$ number of bits. We can represent this message as an n-degree polynomial $M(x)$. Suppose we want to send an 8-bit message 11010101. Its corresponding polynomial representation would be as follows:

$$M(x) = x^7 + x^6 + x^4 + x^2 + 1$$

In order to calculate the CRC of this message, the source and destination node must choose a common divisor polynomial (otherwise known as the generator polynomial) $G(x)$ of degree k. Let us choose $G(x)$ to be $x^3 + x^1 + 1$ with $k = 3$. In binary form, this would be 1011. A well chosen divisor polynomial can lead to detection of many different types of bit errors.

The message string needs to be appended with k number of zeroes before we perform the modulo 2 division operation on the message string. Let us call this zero-padded message string $T(x)$. Now we need to do modulo 2 division of $T(x)$ with $G(x)$ and obtain the remainder. After that, we need to subtract this remainder from $T(x)$. This is the same as appending the remainder to the original message string ($M(x)$), as the addition and subtraction operations are both Exclusive-OR (XOR) in modulo 2 arithmetic. Now we can send the message string, appended with the remainder, to the destination node. The receiver node can do the same modulo 2 division of the received polynomial with $G(x)$ and check the obtained remainder to see if it is zero. The zero remainder shows that no errors have been introduced in the received polynomial. The receiver can otherwise check the integrity of the received message by keeping the remainder (the CRC) aside and calculating the remainder once again by doing modulo 2 division of the zero padded message string $T(x)$. If the calculated remainder matches with the remainder received, then with much higher probability we can infer that the received message is error free.

Let us illustrate the modulo 2 division process with our example message string 11010101. The chosen divisor string is 1011 and the zero-padded message string would be 11010101000.

```
     1111011
    ------------
1011)11010101000
     1011
     -----------
      1100
      1011
      ----------
       1111
       1011
       ---------
        1000
        1011
        --------
         111
         000
         -------
         1110
         1011
         -------
          1010
          1011
          ------
            010
          ------
```

After the modulo 2 division, we get 010 as remainder. If we append this to the message string, we get 11010101010. This is what the sender needs to send to the receiver.

3.2 Why CRC32C?

CRC32C uses the 32-bit generator polynomial (Castagnoli polynomial) $x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^{9} + x^{8} + x^{6} + 1$ [10]. In hexadecimal form, we represent it as 0x1EDC6F41 and the binary representation is as follows:

1 0001 1110 1101 1100 0110 1111 0100 0001

Ignoring the first one from the left, if we reverse the above binary notation, then we get:

1000 0010 1111 0110 0011 1011 0111 1000

and its corresponding hex notation is 0x82F63B78. There exists two other notations as well; reciprocal (0x05EC76F1) and reversed reciprocal (0x8F6E37A0).

The effectiveness of a CRC polynomial for a particular codeword length depends on the minimum Hamming Distance (HD) that it provides among the codewords. The larger the minimum HD at a certain codeword length, the better the CRC polynomial. The IEEE 802.3 CRC32 polynomial (0x04C11DB7) achieves HD=4 up to 91,607 bits and HD=6 up to 268 bits; whereas, CRC32C achieves HD=4 up to 131,072 bits and HD=6 up to 5,243 bits [24]. Therefore, CRC32C has better error detection performance as compared to Ethernet CRC32.

Koopman [24] has recommended the use of 32-bit CRC polynomial 0xBA0DC66B in the newer protocols for significantly improved error detection as it provides HD=4 up to 114,663 bits and HD=6 up to 16,360 bits, and therefore it is much better than IEEE 802.3 CRC and CRC32C. However, modern Intel processors have hardware instructions for CRC32C, and ARM processors have hardware instructions for Ethernet CRC32 and CRC32C. Neither Intel nor ARM processor has hardware accelerated support for computing Koopman CRC32. This is the primary reason why we are recommending the use of CRC32C polynomial in order to perform hardware accelerated CRC32 computation with better end-to-end error detection in TCP. The carry-less multiplication based Koopman CRC32 computation would be more expensive.

CRC has some useful properties listed below:

- CRC is data independent because the relative positioning of bits in the data blocks does not affect the error detection performance unlike the one's complement checksum where the simultaneous flipping of zero and one bits in the same bit position in the data block would lead to an undetected error.

- CRC obeys linearity principle. Mathematically,

$$crc(a \oplus b \oplus c) = crc(a) \oplus crc(b) \oplus crc(c) \qquad \texttt{(XOR)}$$

$$crc(a \cdot b \cdot c) = crc(a) \cdot crc(b) \cdot crc(c) \qquad \texttt{(carry-less multiplication)}$$

- CRC with k-degree generator polynomial ($G(x)$) having non-zero coefficients for the terms x^{k} and x^0 is able to detect all single bit errors.

- CRC with the generator polynomial having a factor with at least three terms is able to detect all double bit errors.

- CRC with the generator polynomial having the factor (x + 1) is able to detect any odd number of errors.

- CRC with k-degree generator polynomial is able to detect any burst error for which the burst length is less than k number of bits. Many burst errors with the burst length greater than k bits can also be detected [29].

The CRC32C polynomial exhibits all of these properties, and the probability of undetected error is much better for CRC32C (approx. 2^{-32}) as compared to the 16-bit one's complement checksum (approx. 2^{-16}). If we consider the packet length to be around the Ethernet MTU size, then CRCs are much more effective at such codeword length as compared to Fletcher and Adler checksums [11].

4. CHECKSUM AND CRC ALGORITHMS

4.1 One's complement checksum

Almost every modern computer uses two's complement to represent the integers. In order to perform one's complement addition on a two's complement system, we need to do end-around carry, i.e., any overflows from the most significant bit need to be added to the least significant bits [18].

One's complement checksum can be incrementally modified. The CRC also could be potentially modified incrementally (because of the linearity property) as stated in RFC 3385 [5]. However, the Internet Experiment Note (IEN) 45 by Bill Plummer, available as extended appendix to RFC 1071 [18], states that CRC cannot be incrementally modified. One's complement addition has some useful properties like commutativity, associativity and byte order independence. As we take the carry bit coming out from the Most Significant bit (MSb) into consideration by means of wrap around and adding back to the Least Significant bit (LSb), the bit flips affecting pair of zeroes and/or pair of ones in the MSb can be detected by one's complement addition based checksum. However, the two's complement addition based checksum would not be able to detect this [18].

In this research, we have considered three versions of the one's complement checksum algorithm which are described here.

4.1.1 One's complement checksum with 16-bit data word

By 16-bit word length, we mean 16-bit worth of data is taken at a time to perform the one's complement addition. The algorithm is really simple indeed. The adjacent bytes upon which the algorithm intends to run are paired to form 16-bit (double byte) words, and then we do one's complement addition of these 16-bit words sequentially until we reach the end of the data buffer.

If we have odd number of bytes in our data buffer, then we need to pad a zero byte at the end to make the total number of bytes in the buffer a multiple of 2. In a C program, this is typically done via typecasting. On a little-endian machine, the buffer can be thought of as an array of 16-bit words of the type uint16_t. At the instant when we are done checksumming all of the 16-bit words except the last remaining byte, the pointer points to this last byte in the buffer. Now we can typecast the buffer to the type uint8_t and add the last byte to the sum. This typecasting is needed to ensure that all the Least Significant Bytes (LSBs) of the 16-bit words are summed together and similarly all the Most Significant Bytes (MSBs) are added together. On a big-endian machine, we do not need this sort of typecasting as the last byte would be automatically added to the MSB of the 16-bit sum.

Actually, in the C program, we store the sum in a 32-bit accumulator. The overflows from the 16-bit one's complement addition is stored in the upper two bytes of this accumulator. At the end of the entire checksumming process, we shift the content in the upper two bytes of the accumulator, and add it to the lower two bytes of this accumulator. There may be an overflow one more time resulting from this addition, and we need to follow the same process of shifting and adding the upper two bytes to the lower two bytes of this accumulator to get the final sum. Finally, the sum (accumulator) is inverted, and this inverted sum (typecasted to uint16_t) is returned as the output of this algorithm.

4.1.2 One's complement checksum with 32-bit data word

Here the consecutive bytes in the data buffer are clustered to form 32-bit (four byte) words. The buffer can be interpreted as an array of 32-bit words of the type uint32_t. We do a one's complement sum of these 32-bit words while the length[1] of the buffer is greater than or equal to four. After each iteration of addition of 32-bit words, we decrement the length of the buffer by four.

Once the remaining length of the buffer falls below four, we check to see if that length is greater than or equal to two. If it is so, then we typecast the buffer to uint16_t type and add the next two bytes directly to the previously calculated sum. Then we further decrement the buffer length by two. If there is still a left out byte that needs to be taken care of, then we typecast the buffer to uint8_t type, and add the last byte to the accumulated sum.

We can use 64-bit accumulator to hold the sum of the 32-bit words and the remaining two

[1]Buffer length is in bytes

byte and/or one byte words present in the buffer. To account for the overflows from the one's complement addition, that is stored in the upper four bytes of the accumulator, we need to add those upper four bytes to the lower four bytes. This operation is called "folding". Once again we may get some overflow bits which we need to add to the least significant bits of the accumulator. Finally, with the upper four bytes cleared off, the lower four bytes again need to be folded to get a 16 bit word and possibly some more carry bits. Every time we get carry bits, we have to perform end-around carry. The 16-bit inverted sum is the final return value of this algorithm.

4.1.3 One's complement checksum with 64-bit data word

Until now, we have not considered adding two 8-byte data words at a time. If we go for adding two such 64-bit data words in a C program, we will loose the carry bit resulting from that addition because we do not have access to the carry flag in the C programming language. Rather we can use x86_64 assembly language, where we can take advantage of the "adc" (add with carry) instruction.

If the length of the data buffer and the CPU info, where the code would be deployed, is known beforehand; then we can precisely optimize the code by exploiting techniques like "loop unrolling" and "code vectorization". Here we describe a generalized version of the algorithm.

First we check if the length of the buffer is greater than or equal to 32 bytes. If the buffer length happens to be greater than 32, we add first 8 bytes of the buffer to the initialized value of the sum (which is a zero for the TCP checksum). Using the adc instruction, we can then add the next sequence of 8 bytes to the previous sum value and so on. We also decrement the length of the buffer by 32 at the end of each iteration. The adc instruction takes into account the carry flag that might have been set by a previous summation operation.

If the buffer length drops below 32, we check if it is greater than or equal to 16. If it is so, then we add the sequentially present next two 8 bytes in the buffer in two steps taking carry bit into consideration, and decrement the buffer length by 16. Then we check if the buffer length is greater than or equal to 8. For the true case, we add the next contiguous 8 bytes to the accumulated sum and decrement the length by 8. We check next if the buffer length is greater than or equal to 4, and if true, we add the next 4 bytes in the buffer to the lower 4 bytes of the accumulated sum in a

64-bit register. Then we decrement the length of the buffer by 4. After that, we check if the buffer length is greater than or equal to 2. If yes, then we add the next 2 bytes in the buffer to the lower 2 bytes of the accumulator. Finally, to see if there is still a byte that is left out, we check if the buffer length is greater than 1. If the condition satisfies, then we add the remaining last one byte in the buffer to the LSB of the accumulator.

Now we need to fold the 64-bit sum that is stored in the accumulator to a 16-bit sum value. This folding process has already been described before. Finally, we invert the 16-bit sum, and return that as the output of this algorithm.

4.2 Fletcher-32 checksum

In a one's complement checksum, we add the consecutive 16-bit words one by one. If those double byte words in the buffer interchange their position because of any hardware/software error, then the one's complement addition does not change as it is insensitive to the positioning of those 16-bit words in the buffer. The Fletcher checksum was introduced by John G. Fletcher to address this particular issue of position sensitiveness of the bytes in the buffer [30].

The 32-bit Fletcher checksum is significantly better than the 16-bit one's complement checksum because of two primary reasons. The first reason being 16-bit one's complement checksum value does not change with the swapping of 16-bit words within the buffer and hence the error would go undetected. However, the Fletcher checksum would be able to detect those transposition of 16-bit words in the buffer. Secondly, the 32-bit Fletcher checksum provides a checksum value occupying 32 bits, and this means that the probability of undetected error (P_{ud}) for any random data would be about $\frac{1}{2^{32}}$. As the checksum size is only 16 bits in case of the 16-bit one's complement checksum, this checksum algorithm has about $\frac{1}{2^{16}}$ probability of not detecting the error which is much higher than that of the Fletcher checksum algorithm.

We have considered only the 32-bit Fletcher checksum in our work, although other variants exist (e.g., Fletcher-16). The computational optimizations suggested by Nakassis [31] have also been incorporated. The implementation details are widely available in the Internet, and we intend to describe here the basic framework of the algorithm. For the 32-bit Fletcher checksum, the data buffer is treated as an array of 16-bit words. Two running sums are computed, which are each of size 16 bits. These sums are accumulated, however, in two 32 bit variables (e.g., `sum0` and `sum1`) that are initialized to zero. The following two summations are central to this algorithm.

```
sum0 = sum0 + *data++; sum1 = sum1 + sum0;
```

We perform the above two additions inside a loop where the loop iterates over 360 times if the buffer length is greater than or equal to 360 number of 16-bit words. However, if the buffer length is less than 360 double byte words, then the above two additions are performed that many times. The rationale behind this is, there will not be any overflow from these two 32-bit accumulators (`sum0` and `sum1`) until 360 iterations. So, after 360 iterations, we need to fold the 32-bit accumulated sum values into 16-bit sums. If we still have more remaining 16-bit words in the buffer, then we need to further execute the loop that many times, and fold the accumulated sums once again and so on. Finally, we need to do the fold operation one more time to ensure that `sum0` and `sum1` do not have any bits left (i.e. the carry) in their upper 16 bits. We return the concatenation of `sum1` and `sum0` as the 32-bit output. This is achieved by left shifting the `sum1` by 16 bits and ORing the result with `sum0`.

If there are N number of 16-bit words in the buffer (zero padding needed for odd buffer length in bytes), then `sum0` would contain `data[0] + data[1] + ... + data[N-1]`, which is nothing but the one's complement addition of the 16-bit data words. The `sum1` would contain `N.data[0] + (N-1).data[1] + ... + data[N-1]`.

4.3 Adler-32 checksum

While the Fletcher-32 checksum performs one's complement addition modulo 65535, the Adler-32 checksum uses a prime modulus (65521) [22]. It is very similar to Fletcher-32 checksum.

In this algorithm, we initialize `sum0` to 1 and `sum1` to 0. The modulo sums that we do in this algorithm are as follows:

$$\texttt{sum0 = (sum0 + *data++)} \bmod 65521\texttt{; sum1 = (sum1 + sum0)} \bmod 65521\texttt{;}$$

Here the data is of type (unsigned char *) and the accumulators (`sum0` and `sum1`) are of type unsigned long. `sum0` basically contains the sum of the bytes present in the buffer and `sum1` contains the running sum of `sum0`. Because of the modulo operation, the `sum0` and `sum1` would never be more than 16 bits. Once those two sums are computed over the entire length of the buffer, we now concatenate `sum1` with `sum0` and return that as output. This concatenation can be done by left shifting the `sum1` by 16 bits and adding it to `sum0`.

The Adler-32 checksum is much better than 16-bit one's complement checksum for the same reasons why the Fletcher-32 checksum is better than 16-bit one's complement checksum as mentioned before. Since the Adler-32 checksum uses a prime modulus, the hope is that it will catch certain patterns of error that the Fletcher-32 checksum would miss otherwise [11].

4.4 Cyclic Redundancy Check (CRC)

The CRC algorithm dates back to the 1960s [32]. There are many tutorials and guides available [33] [34] [35] that have extensively discussed hardware (shift register based) and software implementations of the CRC algorithm. Here we present only those algorithms that we have included in our research work. In each of these algorithms, we have used CRC32C (Castagnoli Polynomial 0x1EDC6F41) and the “reflected bit order” for both the input and the resultant CRC. Also the initial CRC value is 0xFFFFFFFF and the final XOR value is 0xFFFFFFFF. We have used reflected look up table (LUT) in the table driven algorithms.

4.4.1 CRC32C bit-wise

In this algorithm, the first byte in the buffer is bit reflected ($destination[7:0] \leftarrow source[0:7]$), left shifted by 24 bits and XORed with the initial CRC value. Next, within a loop, we check if the most significant bit of the resulting CRC is set. If it is set, then we left shift the CRC by 1 bit and XOR with the CRC32C polynomial. Otherwise, we just left shift the CRC by 1 bit. We continue this checking of CRC most significant bit set condition until we pop out 8 most significant bits of the running CRC. Once we have done checking this, we decrement the length of the buffer and increment the buffer pointer to point to the next byte in the buffer.

Now we again bit reflect this byte, shift it left by 24, XOR it with the running CRC, and then the checking of MSb set condition ensues as described earlier. By the time we cover all the bytes present in the buffer, we would get our final CRC which needs to be bit reflected ($destination[31:0] \leftarrow source[0:31]$). This final bit reflected CRC also needs to be XORed with the final XOR value. As XORing a variable with all 1s is same as inverting the variable, we can go for inverting the final bit reflected CRC instead of doing the XOR. This inverted bit reflected CRC is the final return value of this algorithm.

This algorithm is considered naive; as it only takes into account one bit at a time, does multiple operations just for this one bit and then moves on to the next bit. This algorithm is therefore very inefficient.

4.4.2 CRC32C nibble-wise

This algorithm takes 4 bits of the input data at a time, and uses a reflected look up table having 16 number of 32-bit entries to find the CRC. As the nibble (4-bits) can be anything from 0000 (0x0) to 1111 (0xF), we have total 16 number of entries in the LUT. The entries are basically `CRC[0x00]`, `CRC[0x10]`, `CRC[0x20]`, .., `CRC[0xF0]`. Here we present an example that shows how to calculate the CRC32C[2] of a byte using the nibble-wise algorithm.

```
CRC[0x73] = CRC[(CRC[0x3] ^ 0x7) & 0x0F] ^ (CRC[0x3] >> 4)
```

In the above equation, `CRC[0x3]` is actually mapped to `CRC[0x30]` which is the fourth entry in the look up table used over here.

Using this nibble-wise algorithm, we can significantly improve the speed of calculating the CRC of the input data bytes compared to the bit-wise algorithm. We need 2^2x2^4 = 64 bytes of storage space to hold the look up table, and it is completely worth it given the enhanced error

[2]The terms CRC and CRC32C have been used interchangeably

detection performance that we get with CRC32C.

4.4.3 CRC32C byte-wise

Here we directly take a byte at a time into consideration in order to calculate the intermediate value of CRC. Hence, this algorithm is significantly more computationally efficient as compared to the bit-wise and nibble-wise algorithms. The data buffer is treated as a sequence of bytes. We need to do table look ups to find the CRCs of the individual bytes in the buffer. The table that we have used in our work is the reflected look up table, where each input byte is bit reflected and their corresponding CRCs are also bit reflected. The initial CRC value and the final XOR value are both 0 for each of the entries in the table. This reflected LUT, for the byte-wise algorithm, has 256 entries (each 32-bit wide). So, the table size is 2^2x2^8 = 1 KiB.

The first byte in the buffer is XORed with the LSB of the initial CRC value to get an 8-bit number. This 8-bit value is used as an index to find the corresponding entry in the table. The initial CRC is right shifted by 8. The resulting value is XORed with the reflected CRC value (that we found from the previous table look up) to get the intermediate CRC for our first byte in the buffer.

Then we decrement the buffer length and increment the buffer pointer. Now we XOR the next byte in the buffer with the LSB of the previously calculated intermediate CRC. This gives us the index to look up the corresponding CRC value in the table. The previously calculated intermediate CRC (for the first byte) is right shifted by 8, and then XORed with the CRC (found from the LUT) to give the intermediate CRC. By this step, we have processed two bytes in the buffer. For the subsequent bytes, we need to follow the same routine again. Once we are done with all the bytes in the buffer, the last CRC value that we get needs to be inverted, and is returned as the output of this algorithm. Since we have used the reflected look up table in this byte-wise algorithm, we do not need any further bit reflection of the final CRC value.

4.4.4 CRC32C slicing-by-4

The byte-wise algorithm, even though performs much better when compared to the bit-wise algorithm, is still not that efficient. Because, these days, typical word size in a system is 32 bits or 64 bits. So, if we can process 32 bits or 64 bits at a time, then the performance would improve significantly.

In the slicing-by-4 algorithm, we use a table of size 4x256 with each entry being of size 32 bits. Hence, the total space needed to store the table is 2^2x2^8x2^2 = 4 KiB. This table makes use of the linearity property of the CRC. Let us visualize this. Let us assume a four byte data value be 0x01 0x02 0x03 0x04. Following the linearity principle of CRC,

```
CRC[0x01 0x02 0x03 0x04] =
CRC[0x01 0x00 0x00 0x00] ^
CRC[0x00 0x02 0x00 0x00] ^
CRC[0x00 0x00 0x03 0x00] ^
CRC[0x00 0x00 0x00 0x04]
```

`CRC[0x00 0x00 0x00 0x04]` is same as `CRC[0x04]` as the leading zeroes do not affect the CRC value. Similarly, `CRC[0x00 0x00 0x03 0x00]` is same as `CRC[0x03 0x00]`, and `CRC[0x00 0x02 0x00 0x00]` is equivalent to `CRC[0x02 0x00 0x00]`. We can look up these CRC values in the look up table (LUT) as follows:

```
CRC[0x00 0x00 0x00 0x04] = LUT[0][4]
CRC[0x00 0x00 0x03 0x00] = LUT[1][3]
CRC[0x00 0x02 0x00 0x00] = LUT[2][2]
CRC[0x01 0x00 0x00 0x00] = LUT[3][1]
```

Here, we have denoted the look up table as `LUT[4][256]` which is a two-dimensional array with 4 number of rows (0 to 3) and 256 number of columns (0 to 255). `LUT[0][4]` is row-0 and column-4 entry in the look up table, and it gives the CRC32C value for the byte 0x04. For the byte value 0x03, that is left shifted by 8, we need to find the row-1 and column-3 entry in the look up table in order to calculate the CRC32C. Now, we can observe a pattern in looking up entries in the look up table. For the MSB of a 4 byte entity, we need to look up the row number 3 and use the MSB value as an index to find the corresponding column entry. That entry would give us the

CRC32C value for the MSB of a 4 byte number. Similarly, for the other bytes of a 4 byte entity, we can find their CRC32C values by appropriately referring to the look up table.

In order to calculate the final CRC32C value for the entire buffer, we need to treat the buffer as an array of 4 byte entities. If the buffer length is not a multiple of 4, then the remaining 1 to 3 bytes can be taken care of by the byte-wise CRC32C algorithm. To begin with, first we take the starting 4 bytes of the buffer, and then use the look up table as described above to find the CRC. Then we increment the buffer pointer to point to the next 4 bytes and XOR it with the CRC value (calculated before) for the first 4 bytes in the buffer. The resulting 4 byte value is split into 4 parts so as to index into the look up table appropriately and calculate the CRC. By this step, we have finished processing the first 8 bytes in the buffer. This is how we can continue processing the next bytes present in the buffer. Once we are done calculating the final CRC, we need to invert it and return as the output of the slicing-by-4 algorithm.

4.4.5 CRC32C slicing-by-8

To increase the computational efficiency even further, we can treat the buffer as an array of 8 byte values. If the buffer size is not a multiple of 8, then the remaining bytes could be handled using the standard byte-wise CRC32C algorithm. The slicing-by-8 algorithm is very similar to slicing-by-4. In slicing-by-8, we use a look up table of size 8x256 with each entry being 32 bits wide. The total space that the look up table occupies would be 2^3x2^8x2^2 = 8 KiB. For a sample 8 byte data, 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08, we can calculate the CRC following the linearity principle of CRC as follows:

```
CRC[0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08] =
CRC[0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00] ^
CRC[0x00 0x02 0x00 0x00 0x00 0x00 0x00 0x00] ^
CRC[0x00 0x00 0x03 0x00 0x00 0x00 0x00 0x00] ^
CRC[0x00 0x00 0x00 0x04 0x00 0x00 0x00 0x00] ^
CRC[0x00 0x00 0x00 0x00 0x05 0x00 0x00 0x00] ^
CRC[0x00 0x00 0x00 0x00 0x00 0x06 0x00 0x00] ^
```

```
          CRC[0x00 0x00 0x00 0x00 0x00 0x00 0x07 0x00] ^
          CRC[0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x08]
```

We can represent the look up table as `LUT[8][256]`, and use the byte values to index into the look up table as follows:

```
CRC[0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x08] = LUT[0][8]
CRC[0x00 0x00 0x00 0x00 0x00 0x00 0x07 0x00] = LUT[1][7]
CRC[0x00 0x00 0x00 0x00 0x00 0x06 0x00 0x00] = LUT[2][6]
CRC[0x00 0x00 0x00 0x00 0x05 0x00 0x00 0x00] = LUT[3][5]
CRC[0x00 0x00 0x00 0x04 0x00 0x00 0x00 0x00] = LUT[4][4]
CRC[0x00 0x00 0x03 0x00 0x00 0x00 0x00 0x00] = LUT[5][3]
CRC[0x00 0x02 0x00 0x00 0x00 0x00 0x00 0x00] = LUT[6][2]
CRC[0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00] = LUT[7][1]
```

The pattern similar to that used in slicing-by-4 is employed here to look up the CRC value of a byte. The index value depends on the position of the byte within the 8 byte entity and the byte value as well.

Once we are done handling all the 8 byte entities and possibly some left out bytes present in the buffer, we will get the final CRC value. Then we need to XOR it with 0xFFFFFFFF (same as inverting) and the resulting value is what we return as the output of this algorithm.

4.4.6 CRC32C using Intel Intrinsics

Intel Intrinsics provide C++ application programming interfaces (APIs) which use assembly instructions at their core [36]. These intrinsics are also expanded inline and therefore, the function call overhead is eliminated. They improve the code readability, as the programmer does not directly code the assembly instructions. SSE4.2 instruction set added the CRC32 instruction for the very first time [37]. On an Intel system, we can check for the presence of CRC32 instruction by executing the CPUID instruction with EAX = 1, and after that we can check to see if the bit 20 of ECX is set or not.

The CRC32 instruction takes two operands. The destination operand holds the initial CRC value and the source operand points to the buffer on which the CRC32C value needs to be calculated. The computed CRC value is stored in the destination operand. While the source operand can be a memory location or a register, the destination operand needs to be a 32-bit or a 64-bit register. We have 8-bit, 16-bit, 32-bit and 64-bit variants of the hardware CRC32 instruction that uses the Castagnoli Polynomial 0x1EDC6F41. Further details of the CRC32 instruction can be found in the SSE4 programming reference [7].

The algorithm that computes CRC32C with Intel Intrinsics uses the following built-in functions: _mm_crc32_u64, _mm_crc32_u32, _mm_crc32_u16 and _mm_crc32_u8. First, we try to divide the input data buffer into 8 byte chunks, and use _mm_crc32_u64 in a loop to calculate the CRC of 8 bytes of data at a time. We accordingly increment the buffer pointer as well. We can then use _mm_crc32_u32 (if the buffer length is greater than or equal to 4) to calculate the CRC of the next 4 bytes of data. If the remaining length of the buffer is still greater than or equal to 2, then we use _mm_crc32_u16 to take care of the next 2 bytes of data. The last but one byte can be taken care of by the built-in function _mm_crc32_u8, and we get our final CRC value. This final CRC value needs to be inverted and is returned as the output of this algorithm.

4.4.7 CRC32C using PCLMULQDQ and CRC32 instructions on Intel

The white paper [37] by Gopal et al. provides a detailed discussion of the implementation of the CRC32C algorithm on Intel processors using the hardware instructions CRC32 and PCLMULQDQ. Here we try to present the key features of this algorithm.

The algorithm cleverly uses the fact that "CRC32 instruction has a latency of 3 clock cycles and a throughput of 1 clock cycle." This means, if we use the CRC32 instruction to calculate the CRC32C of the data byte-by-byte or multi byte word-by-word, then the subsequent bytes or words in the buffer have to wait for at least 3 clock cycles in order for the CRC computation on the previous byte or word to finish first. This CRC32 instruction has been described in the previous algorithm.

We divide the buffer into three parts such that we can process 8 bytes of data at a time, starting from these three offsets, in parallel. This parallelized CRC computation is done to improve the overall throughput of the process. Once we get the individual CRC values of these three segments, we need to recombine these CRCs to get a resultant CRC.

In this algorithm, the recombination step is done using the PCLMULQDQ instruction. As we calculate the CRC values of the three parts of the buffer beginning at three different offsets, we get the CRC values at three different offsets as well. The CRC values of the first two parts of the buffer need to be shifted to the right so as to align with the CRC value of the last part of the buffer. Using the carry less multiplication, we can shift those 2 CRC values to the right. In addition to shifting the CRC values to the right, the carry less multiplication also increases the size of shifted CRC values to 64 bits. So, we need to calculate again the CRC of each of these shifted values, and then XOR the results together with the CRC value calculated on the last part of the buffer that is already there in its correct position. This is how we obtain the resultant CRC of those three parts of the buffer via the recombination step.

If we still have some remaining bytes in the buffer that have not been processed, then we can start processing the rest of the buffer 8 bytes at a time. Once the remaining buffer length becomes smaller than 8 bytes, we can process the next 4 bytes of data at a time (if the buffer length is greater than or equal to 4). The remaining 2 bytes and/ or 1 byte of data can be taken care of by the 16-bit and/ or 8-bit version of the CRC32 instruction. The final CRC value needs to be inverted, and is returned as the output of this algorithm.

4.4.8 CRC32C using CRC32 instruction on Intel and table-based recombination

Mark Adler has provided a code (specific to Intel CPUs supporting SSE4.2 CRC32 instruction) on Stack Overflow that computes the CRC32C of the input data buffer using the hardware CRC32 instruction [38]. Fundamentally, this code is very similar to the previous algorithm where we divide the buffer into 3 parts, calculate the CRCs independently, and recombine them at the end to obtain the resultant CRC.

The code provided by Adler also divides the input data into three large parts, and calculates the CRCs of those three parts independently using the 64-bit variant of the hardware CRC32 instruction (crc32q). However, this code recombines those CRCs with differing offsets via the table look up method instead of using the PCLMULQDQ (carry less multiplication) instruction. The size of this look up table is 2^2x2^8x2^2 = 4 KiB. This look up table holds the CRCs for the bytes that have been appropriately left shifted by a fixed large value. The CRC value that we calculate for the first part of the buffer (let us say crc0) and for the second part of the buffer (let us say crc1), needs to be aligned with the CRC value obtained from the third part of the buffer (let us say crc2). To achieve this, the code first aligns crc0 with crc1. This is done by segregating the four bytes that make up crc0, looking up the CRC values corresponding to these bytes, and XORing them together. After this step, whatever we get aligns perfectly with crc1 and hence can be XORed directly with crc1. At this point, whatever 32-bit value we have now needs to be aligned with crc2 and we can follow the same procedure again. At the end, we get the final CRC value of those three parts of the buffer taken together.

This code also tries to divide the buffer into three small parts if the remaining buffer length is not suitably large. The final CRC value obtained in the last step, as described above, is considered as the initial CRC value here. Then, the code calculates the CRCs of these three small parts of the buffer independently, and recombines them at the end to get the resultant CRC of those three small parts taken together. To achieve this, it uses one more look up table of size 2^2x2^8x2^2 = 4 KiB. So, in total, Adler's code uses 8 KiB worth of space to store the LUTs. The look up table used over here carries the CRCs of the bytes that have been left shifted by a fixed small value. We can calculate the CRCs of the three individual small parts of the buffer, and recombine them at the end just like the way we did for the three larger parts of the buffer.

After all these steps, if we are still left with some data bytes in the buffer, then the CRC is calculated in a linear fashion (8-byte at a time, and then 1-byte a time). The final CRC value obtained is inverted, and is considered as the return value of this algorithm.

5. RESEARCH METHODOLOGY

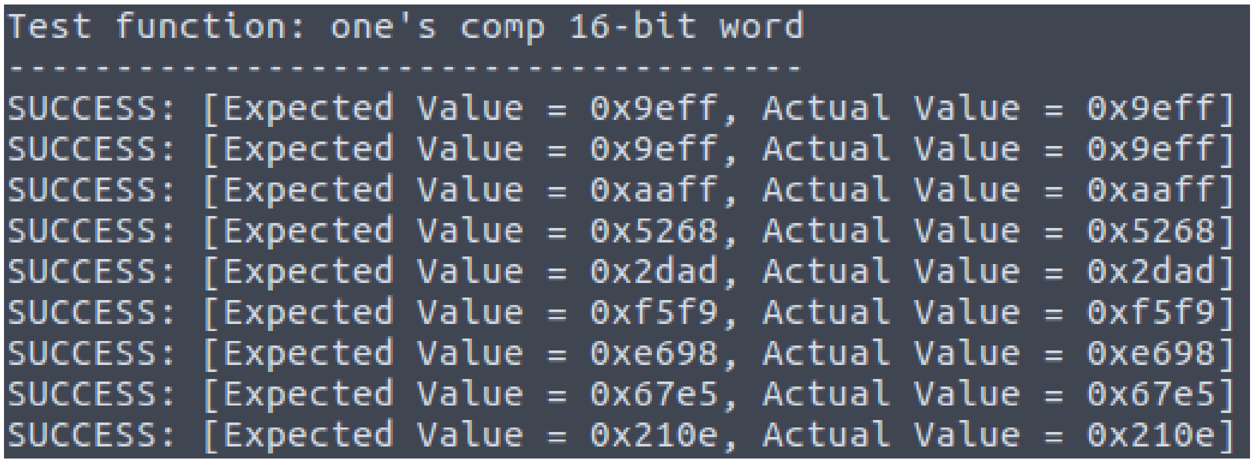
Our objective is to benchmark the variety of checksum and CRC algorithms found in the literature and to make a recommendation for improved end-to-end error detection in TCP using the algorithm that best meets the needs of both the source and destination node.

First, we tried to analyze how all the different checksum and CRC algorithms work. Many of the CRC algorithm implementations found in the Internet have not taken the reflected bit order and the reflected CRC into account correctly [39] [40] [41]. Those implementations did not pass the test routine that we have created with specific use cases. We have referred to many GitHub repositories available in the Internet to get a fair idea about the standard benchmarking practices [42] [43] [44].

We used C and Assembly language programs to test the checksum and CRC implementations. All the codes have been tested on 64-bit platforms only. The compilers and assemblers that we used are gcc, nasm and yasm. We did not program any new checksum algorithms. Rather, we referred to several of the checksum implementations available in the literature (open source) since these algorithms are well known in the area of communications and networking. Wherever we found errors or inconsistencies with respect to these implementations, we made necessary changes so as to properly calculate the iSCSI CRC32C. We developed a custom benchmark routine that calculated the number of CPU cycles per byte that each of the checksum and CRC algorithms require for execution. This benchmark method is targeted to meet the objective that we have set at the very beginning, and hence it takes into consideration some specific buffer sizes of interest and some selected algorithms.

The execution time of the CRC algorithm is independent of the data because the data word values do not affect the CRC computation time. In contrast, the execution time of the one's complement addition-based checksum algorithms is dependent on the data, e.g., the 16-bit TCP checksum has to implement end-around carry whenever there is a carry from the most significant bit. So, if we get such overflow (carry) while doing the one's complement addition of certain data word values, then more execution time would be required. Hence, we chose to use random source of data like /dev/urandom which serves as pseudorandom number generators (PRNG). This random data served as the input buffer for the algorithms under consideration. In each iteration inside a loop, we read a fixed number of bytes (e.g., 64[1], 128, 256, .., 1500[2], .., 9000[3], .., 65536) from the file /dev/urandom, and stored them in a buffer. Then, we passed the buffer pointer, the length of the buffer and the initial value of the checksum (or CRC) to the checksum or CRC algorithm that is currently being tested.

To check the correctness of an algorithm, we created test cases where we provided some fixed-length specific sequence of bytes as input, and we also supplied the expected checksum or CRC value for these sequence of bytes. We calculated this expected checksum value by hand, and we obtained the expected CRC value using the reference [45]. Our test routine called each of the checksum and CRC algorithms, and ran these test cases for each of them. The success/failure information was printed on the console.

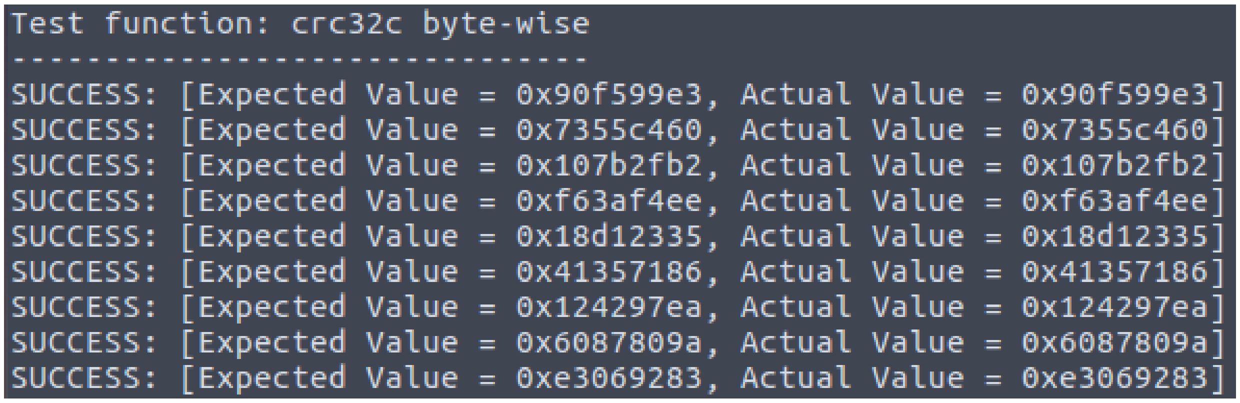


```
Test function: one's comp 16-bit word
-----------------------------------
SUCCESS: [Expected Value = 0x9eff, Actual Value = 0x9eff]
SUCCESS: [Expected Value = 0x9eff, Actual Value = 0x9eff]
SUCCESS: [Expected Value = 0xaaff, Actual Value = 0xaaff]
SUCCESS: [Expected Value = 0x5268, Actual Value = 0x5268]
SUCCESS: [Expected Value = 0x2dad, Actual Value = 0x2dad]
SUCCESS: [Expected Value = 0xf5f9, Actual Value = 0xf5f9]
SUCCESS: [Expected Value = 0xe698, Actual Value = 0xe698]
SUCCESS: [Expected Value = 0x67e5, Actual Value = 0x67e5]
SUCCESS: [Expected Value = 0x210e, Actual Value = 0x210e]
```

Figure 5.1: One's complement checksum test case

In our benchmark routine, we did not change any CPU scheduler parameters. Also, we did not

[1]Representative of ACK only frame payload size
[2]Standard Ethernet MTU
[3]Jumbo frame payload size



```
Test function: crc32c byte-wise
-------------------------------
SUCCESS: [Expected Value = 0x90f599e3, Actual Value = 0x90f599e3]
SUCCESS: [Expected Value = 0x7355c460, Actual Value = 0x7355c460]
SUCCESS: [Expected Value = 0x107b2fb2, Actual Value = 0x107b2fb2]
SUCCESS: [Expected Value = 0xf63af4ee, Actual Value = 0xf63af4ee]
SUCCESS: [Expected Value = 0x18d12335, Actual Value = 0x18d12335]
SUCCESS: [Expected Value = 0x41357186, Actual Value = 0x41357186]
SUCCESS: [Expected Value = 0x124297ea, Actual Value = 0x124297ea]
SUCCESS: [Expected Value = 0x6087809a, Actual Value = 0x6087809a]
SUCCESS: [Expected Value = 0xe3069283, Actual Value = 0xe3069283]
```

Figure 5.2: CRC32C test case

change the default scheduling priority of the process. Furthermore, we did not alter the default scheduling policy. Though the Linux kernel has provisions to specify the CPU affinity sets for a process, we did not modify that either. We could have suitably changed all these parameters to get better results. But we did not do that just to get the worst case (upper bound) values for our benchmark. The algorithms that perform better in such an unprivileged scenario are expected to perform better in a privileged scenario of running them as a kernel-level thread.

We ran our benchmark routine on Intel (Intel Core i3-2330M and Intel Xeon), ARM (Raspberry Pi 3 Model B) and PowerPC (POWER7) systems. We made use of the rdtsc instruction for the benchmark study on the Intel processor. The rdtsc (read time stamp counter) instruction is used to read the current time-stamp counter variable, which is a 64-bit variable, into the registers (edx:eax) [46]. The TSC (time stamp counter) is incremented every CPU tick. For example at 1MHz CPU, the TSC is incremented by 10^6 per second.

We read a certain number of bytes from our random data source in each iteration with an exponentially increasing step size, stored the data bytes in a buffer, and then recorded the TSC value before and after the checksum or CRC calculation on that buffer. This is how we were able to get a good approximation of the number of cycles consumed by the actual running checksum or CRC algorithm. Dividing that by the number of bytes in the buffer gave us our desired performance metric which is "number of CPU cycles per byte." Based on this criterion, we compared different software and hardware instruction implementations of the checksum or CRC algorithms of interest.

For each buffer size, we ran the CRC or checksum algorithms 1000 times each. We recorded the average number of CPU cycles per byte that these algorithms took for all those different buffer lengths. We used this information to build tables and plot graphs in Chapter 6. Based on the standard deviation data, we also calculated the 95% confidence interval for these average number of CPU cycles per byte values that we got for different algorithms at different buffer sizes. The following figure shows that with 95% confidence, the average number of CPU cycles per byte for the CRC32C (CRC32 + PCLMULQDQ) algorithm with 64 bytes buffer size is between 1.2 and 1.26 based on 1000 samples. Similarly, we can see that the average number of CPU cycles per byte for the TCP checksum (64-bit word version) with 1500 bytes buffer size varies between 0.248 and 0.248 (sample mean is 0.248, and margin of error is 0.000242) with 95% confidence.

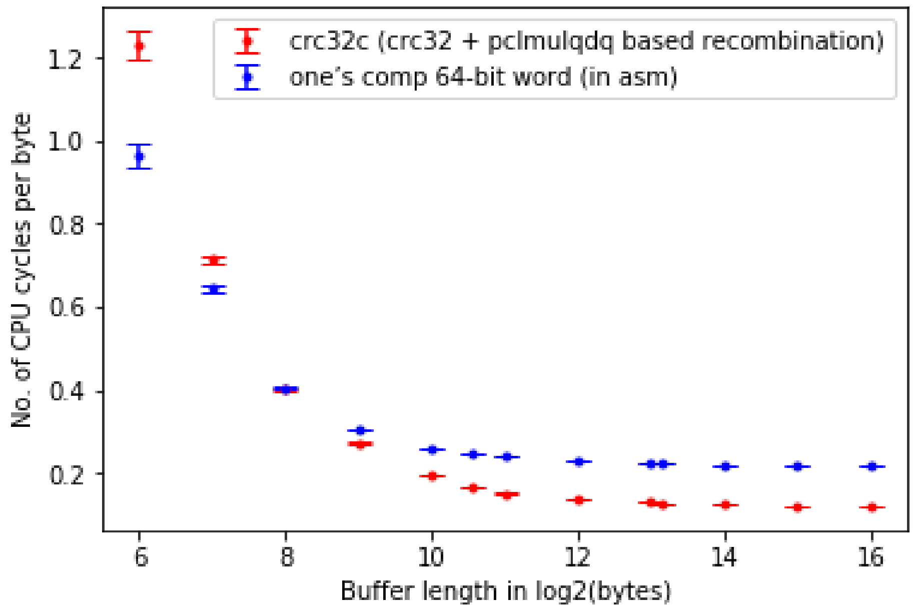


Figure 5.3: Error bars showing 95% confidence interval for the average number of CPU cycles per byte based on 1000 samples for CRC32C (using CRC32 and PCLMULQDQ instructions) and TCP checksum with 64-bit word version on Intel Xeon CPU

On ARM system, we used cntvct_el0 (Counter-timer Virtual Count register) to get the 64-bit virtual count value. Then we used the mrs instruction to move the content of this 64-bit register into a local variable which was later accessed in our program. This count value was recorded before and after the checksum or CRC calculation on the buffer in order to know how many CPU cycles were approximately used by the algorithm itself to run. For each buffer size, we ran all the CRC or checksum algorithms 1000 times each, and then we took the mean of these 1000 number of cycles per byte values to get the average number of CPU cycles per byte that these algorithms took for all the different buffer lengths.

On the PowerPC system, the Time Base (TB) is a 64-bit register whose value is incremented periodically. It provides a long period counter value. We used the mftb instruction to read the Time Base, and stored it in a local variable. This Time Base register value was recorded before and after the checksum or CRC calculation to know the approximate number of CPU cycles used by the algorithm alone to execute. We ran all the CRC or checksum algorithms 1000 times each for all the buffer sizes in order to get the average number of CPU cycles per byte used by different algorithms at different buffer lengths.

For all the buffer sizes of interest, we also injected some extra bytes into the buffer (e.g., adding one extra byte to a 64 byte buffer) to have some bytes in the memory that does not align nicely with the word boundaries (word, double-word or quad-word), and we calculated the CPU cycles per byte for the different checksum and CRC algorithms with these odd buffer sizes.

6.1 Benchmark study of checksum/ CRC algorithms on Intel CPUs

The following graph shows the comparative computational performance of different CRC32C and checksum algorithms discussed in Chapter 4. The processor model used for this evaluation is an Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz running Ubuntu 16.04 LTS, which supports both CRC32 and PCLMULQDQ instructions. A table showing the number of CPU cycles used per data byte (over different data buffer sizes) for different CRC32C and checksum implementations is given in Appendix A.1.

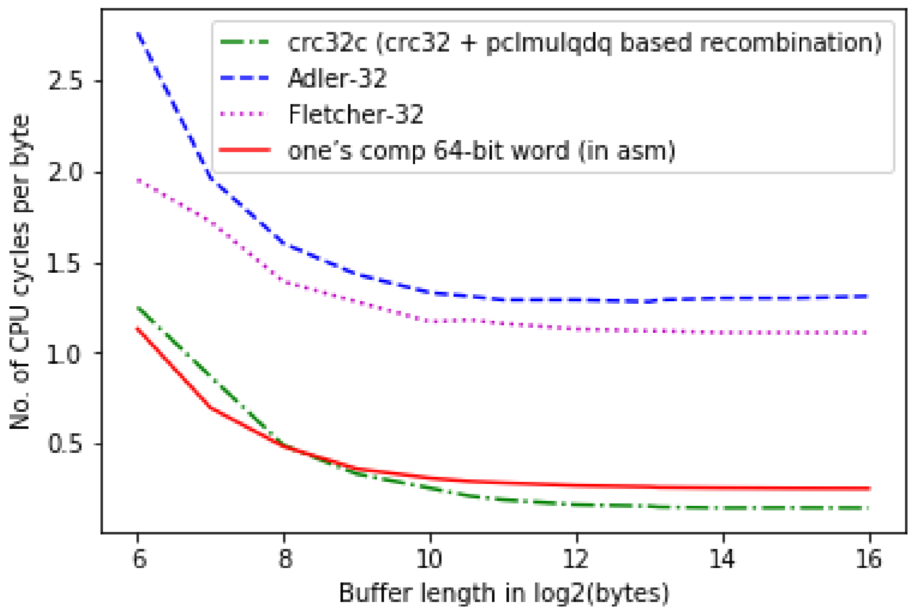


Figure 6.1: Cycles/byte performance of CRC32C using Intel CRC32 and PCLMULQDQ instructions, Adler-32 checksum, Fletcher-32 checksum, and TCP one's complement checksum on Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

Figure 6.1 shows that the CRC32C computation using CRC32 and PCLMULQDQ instructions is 38% faster than TCP one's complement checksum at 1500 bytes, and 72% faster at a buffer size of 9000 bytes. When the buffer length is small (e.g., 64 bytes), the TCP one's complement checksum is 11% faster than CRC32C.

Figure 6.2 shows the same performance measurement done on an Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz running Linux kernel 2.6.32-754.14.2.el6.x86_64. This system also supports both the CRC32 and PCLMULQDQ instructions.

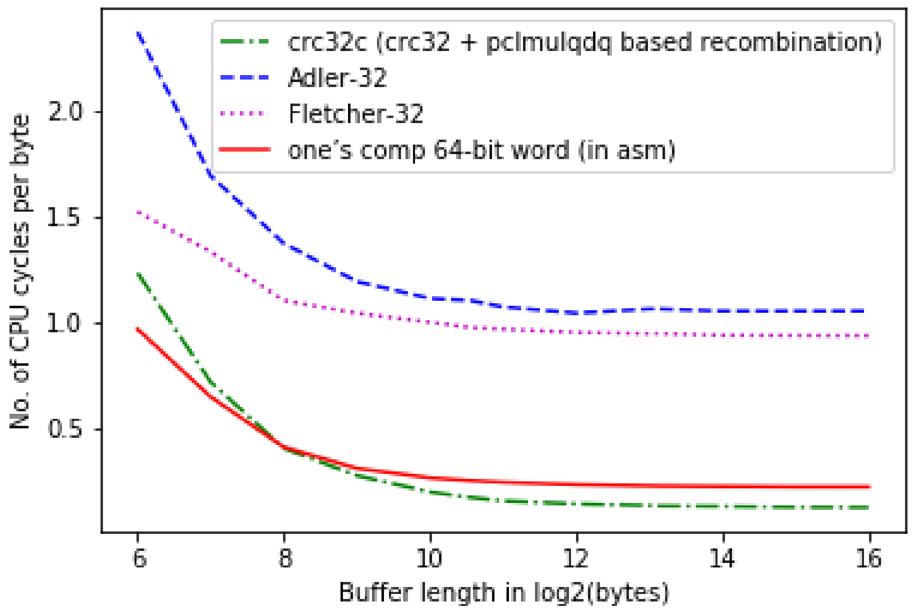


Figure 6.2: Cycles/byte performance of CRC32C using Intel CRC32 and PCLMULQDQ instructions, Adler-32 checksum, Fletcher-32 checksum, and TCP one's complement checksum on Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz

The Adler-32 and Fletcher-32 checksums are slower than CRC32C and the TCP one's complement checksum. This was found to be true for both the Intel CPUs. We can see in the above graph that CRC32C is 48% faster than TCP one's complement checksum at 1500 bytes, and 73% faster for a 9000 byte buffer size. When the buffer length is small (e.g., 64 bytes), the TCP one's complement checksum is 28% faster than CRC32C. We also found that the cycles per byte values for the Intel Xeon CPU are better when compared to the Intel Core i3-2330M across all buffer sizes and for all the checksum/ CRC algorithms that we have considered in our research.

Gopal et al. in an Intel white paper [37] report that for large buffer sizes (greater than 1024 bytes), the CRC32C computation using hardware instructions CRC32 and PCLMULQDQ takes approximately 1.16-1.2 cycles/Qword, which translates to 0.145-0.15 cycles/byte. They have also computed the CRC32C with hardware instruction CRC32 and table-based recombination, and for a fixed buffer size of 1024 bytes, they found the resulting cycles/Qword value to be marginally better than that of CRC32C with PCLMULQDQ based recombination. On the Intel Core i3-2330M, we found that CRC32C calculation using CRC32 and PCLMULQDQ takes approximately 0.15 cycles/byte for buffer sizes greater than or equal to 8192 bytes, which matches Gopal's result for the higher range. For 1024 bytes, the CRC32C calculation using CRC32 and PCLMULQDQ takes approximately 0.26 cycles/byte on the Intel Core i3-2330M. Also we found that, CRC32C with CRC32 and table-based recombination performs better than PCLMULQDQ-based recombination for 64 and 128 byte buffer sizes. For buffer sizes greater than or equal to 256 bytes, CRC32C computation with CRC32 instruction and PCLMULQDQ-based recombination method is faster than that with table-based recombination method.

In another Intel white paper [47], Gopal et al. have tabulated the results for different variants of the iSCSI CRC32C computation with 512, 1024, 2048, and 4096 byte buffers. They have reported that the CRC32C calculation using PCLMULQDQ-based recombination takes 0.15 cycles/byte for a 1024 byte buffer, the CRC32C with table-based recombination takes 0.14 cycles/byte for a 1024 byte buffer, and a serial implementation of CRC32C takes 0.38 cycles/byte for a 1024 byte buffer. Gueron [28] has reported that CRC32C computation with the division of buffer into three parts followed by table-based or PCLMULQDQ based recombination is almost 2.5 times faster than the linear approach (8 byte-by-8 byte, then byte-by-byte) of computing CRC32C. Kounavis et al. [26] have reported that the slicing-by-8 CRC32C algorithm takes 2.39 cycles/byte for large buffer size, when the data buffer and the table are initially warm. Our results are in close agreement with the results from these papers. On the Intel Core i3-2330M for a buffer size of 1024 bytes, we found that the CRC32C calculation with CRC32 and PCLMULQDQ instruction takes 0.255 cycles/byte, the CRC32C with table-based recombination takes 0.355 cycles/byte, and a serial implementation of CRC32C (using Intel Intrinsic) takes 0.554 cycles/byte. We also found that the CRC32C using hardware instructions CRC32 and PCLMULQDQ is almost 2.5 times faster in comparison to the CRC32C using Intel Intrinsic for a 1500 byte buffer size. The slicing-by-8 CRC32C algorithm takes approximately 2.27 cycles/byte for a 1500 byte buffer size as can be seen in Fig. 6.6.

Table 6.1 presents a side-by-side comparative view of the performance of different checksum and CRC algorithms in terms of the number of CPU cycles/byte taken by these algorithms on both the Intel CPUs. We have used GCC optimization flag O3 to get these cycles/byte values for different algorithms. Three different buffer sizes of interest were considered (64, 1500, and 9000 bytes). We chose these specific buffer sizes because 64 byte is a representative size for TCP ACK-only frame; whereas, 1500 byte and 9000 byte buffer sizes represent the standard Ethernet MTU and Jumbo frame payload sizes respectively. On both the Intel CPUs, the CRC32C using hardware instructions CRC32 and PCLMULQDQ outperformed the TCP one's complement checksum for the buffer sizes greater than or equal to 256 bytes.

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:	
		Intel Core i3-2330M	Intel Xeon
crc32c bit-wise	64	160	109
	1500	121	101
	9000	120	101
crc32c nibble-wise	64	17.8	14.9
	1500	17.3	14.3
	9000	17.3	14
crc32c byte-wise	64	10.4	7.8
	1500	9.45	7.03
	9000	9.43	7.01
crc32c slicing-by-4	64	4.5	3.53
	1500	3.51	2.61
	9000	3.47	2.56
crc32c slicing-by-8	64	3.14	2.66
	1500	2.27	1.72
	9000	2.21	1.65
crc32c Intel Intrinsic	64	1.23	1.04
	1500	0.538	0.462
	9000	0.509	0.437
crc32c (crc32 + pclmulqdq based recombination)	64	1.25	1.23
	1500	0.211	0.168
	9000	0.151	0.128
crc32c (crc32 + table-based recombination)	64	1.15	1.03
	1500	0.37	0.279
	9000	0.202	0.17
Adler-32	64	2.76	2.37
	1500	1.31	1.1
	9000	1.29	1.06
Fletcher-32	64	1.95	1.52
	1500	1.18	0.972
	9000	1.12	0.944
one's comp 16-bit word (in C)	64	1.29	1.04
	1500	0.275	0.227
	9000	0.207	0.174
one's comp 32-bit word (in C)	64	1.38	1.04
	1500	0.285	0.23
	9000	0.214	0.174
one's comp 64-bit word (in x86_64 assembly)	64	1.13	0.964
	1500	0.292	0.248
	9000	0.259	0.222

Table 6.1: Comparative computational performance of different CRC32C and Checksum implementations on Intel CPUs

The 16-bit word and 32-bit word variants of the one's complement checksum took an almost identical number of CPU cycles per byte for the buffer sizes greater than or equal to 1024 bytes as can be seen in Fig. 6.3. Surprisingly, we observed that the computational performance of both the 16-bit word and 32-bit word variants of the one's complement checksum surpassed that of the 64-bit word variant of the TCP checksum for buffer sizes greater than or equal to 1024 bytes.

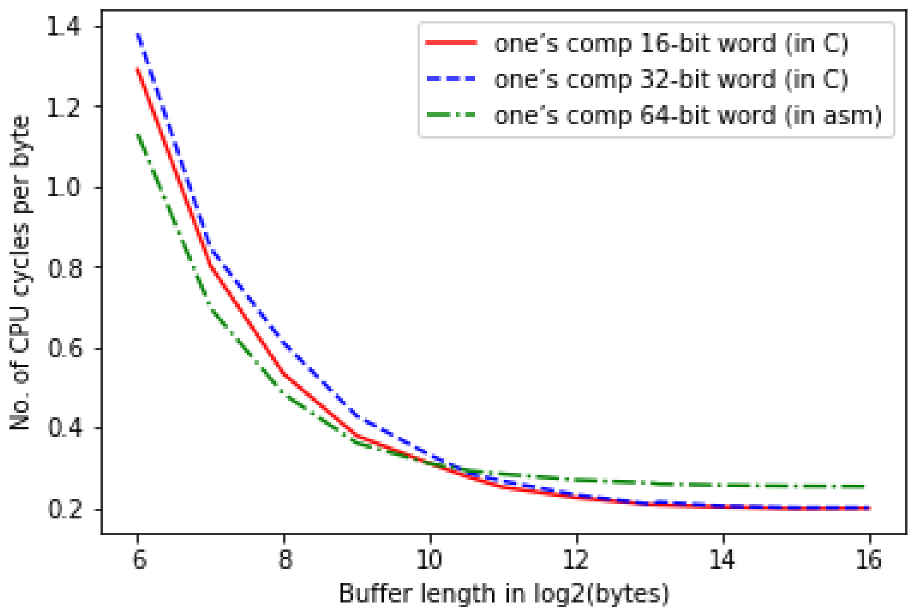


Figure 6.3: Cycles/byte performance of 16-bit word, 32-bit word, and 64-bit word variants of the TCP checksum on Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz with GCC O3 optimization flag

After seeing the above results, which were contrary to our expectation (and the results in [48] described later) that the 64-bit word version of the one's complement checksum would perform better than the 16-bit and 32-bit word variants, we tried changing the GCC optimization flag from O3 to O1. With GCC O1 optimization, we got the results that we had initially expected which are shown in Fig. 6.4. The 64-bit word version of one's complement checksum is approximately 2.5 times faster than the 16-bit word variant and approximately 1.3 times faster than the 32-bit word variant for buffer sizes greater than or equal to 1024 bytes.

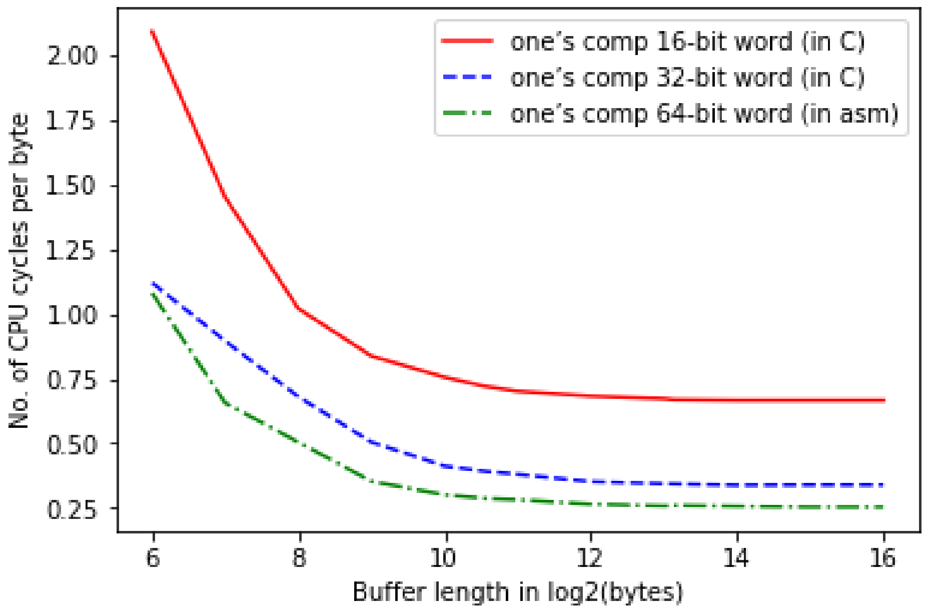


Figure 6.4: Cycles/byte performance of 16-bit word, 32-bit word, and 64-bit word variants of the TCP checksum on Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz with GCC O1 optimization flag

Lockless Inc. has published an article where they have calculated the time taken by different versions of the one's complement checksum algorithm for different buffer sizes [48]. They have reported that the 16-bit word version of the TCP checksum takes approximately 0.819 ns/byte for a 64 byte buffer and 0.642 ns/byte for a 1024 byte buffer; whereas, the 64-bit word version takes approximately 0.27 ns/byte for a 64 byte buffer and 0.17 ns/byte for a 1024 byte buffer [48]. Garcia [49] in a blog titled "Fast checksum computation" has reported results for 64-bit word assembly language version of the one's complement checksum algorithm. The ns/byte values (translated from the cycles/byte values) that we obtained for the different versions of the one's complement checksum (with GCC optimization flag O1) are in close agreement with these results. From the table in A.4, we can see that the 16-bit word variant of the one's complement checksum takes approximately 0.95 ns/byte for a 64 byte buffer and 0.34 ns/byte for a 1024 byte buffer; whereas, the 64-bit word version takes approximately 0.49 ns/byte for a 64 byte buffer and 0.14 ns/byte for a 1024 byte buffer on Intel Core i3-2330M. We can also see from the table that the 16-bit and 32-bit word versions of the one's complement checksum (implemented in C) performed better with

GCC O3 optimization flag than O1. However, the performance of 64-bit word version of the one's complement checksum (implemented in x86_64 assembly) is almost the same with both the O3 and O1 flags.

Figure 6.5 shows the comparative performance of the hardware instruction based CRC algorithms. The CRC32C is computed in a linear fashion (8 byte-by-8 byte, then 4 byte-by-4 byte, followed by byte-by-byte) without any recombination overhead using the Intel Intrinsic. However, the version using CRC32 and PCLMULQDQ instructions explicitly divides the buffer into three parts in order to judiciously use the three clock cycle latency of the CRC32 instruction. From Fig. 6.5 and table 6.1, we can see that the CRC32C using hardware instructions CRC32 and PCLMULQDQ is approximately 2.5 times faster in comparison to the CRC32C using Intel Intrinsic for a 1500 byte buffer size. Gueron [28] has similarly reported that the CRC32C algorithm with CRC32 and PCLMULQDQ instructions achieves almost a speed up of factor 3 as compared to the serial computation of CRC32C for a sufficiently large buffer size.

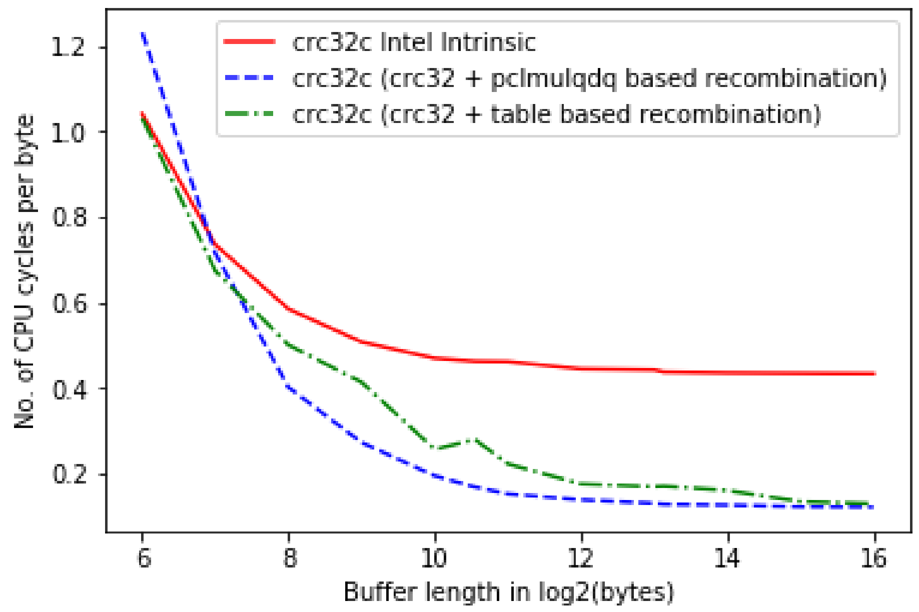


Figure 6.5: Cycles/byte performance of CRC32C using Intel Intrinsic, CRC32 and PCLMULQDQ instructions, only CRC32 instruction on Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz

Figure 6.6 shows how the software-based table-driven CRC32C algorithms (byte-wise, slicing-by-4, and slicing-by-8) performed in comparison to the hardware instruction based CRC32C. As expected, the CRC32C byte-wise algorithm was the slowest followed by slicing-by-4 and slicing-by-8. For large buffer size, the slicing-by-4 CRC32C is approximately 2.7 times faster than the byte-wise algorithm, and the slicing-by-8 CRC32C is approximately 4.3 times faster than the byte-wise algorithm. The slicing-by-8 CRC32C algorithm takes approximately 2.27 cycles/byte for 1500 byte buffer size as can be seen in Fig. 6.6.

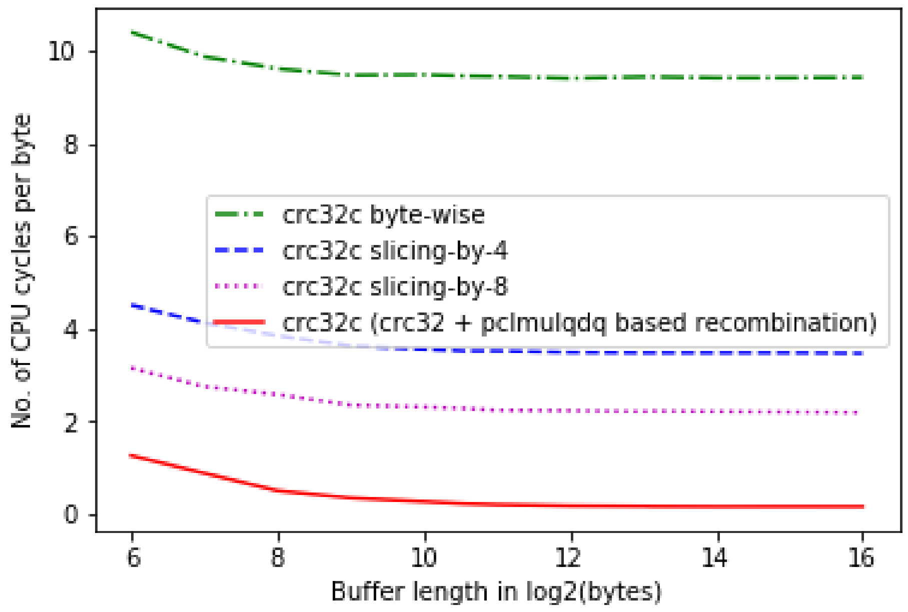


Figure 6.6: Performance comparison of table-driven CRC32C and hardware instruction based CRC32C on Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

The naive bit-wise algorithm performed poorly, as expected, in comparison to the byte-wise, slicing-by-4, and slicing-by-8 CRC32C algorithms, as is evident in Fig. 6.7.

Also, we added a few extra bytes to make the buffer sizes odd, and we computed the number of CPU cycles/byte for all the different checksum and CRC algorithms with these odd buffer sizes. In the Appendix, we have given tables specifically for CRC32C using hardware instructions (CRC32

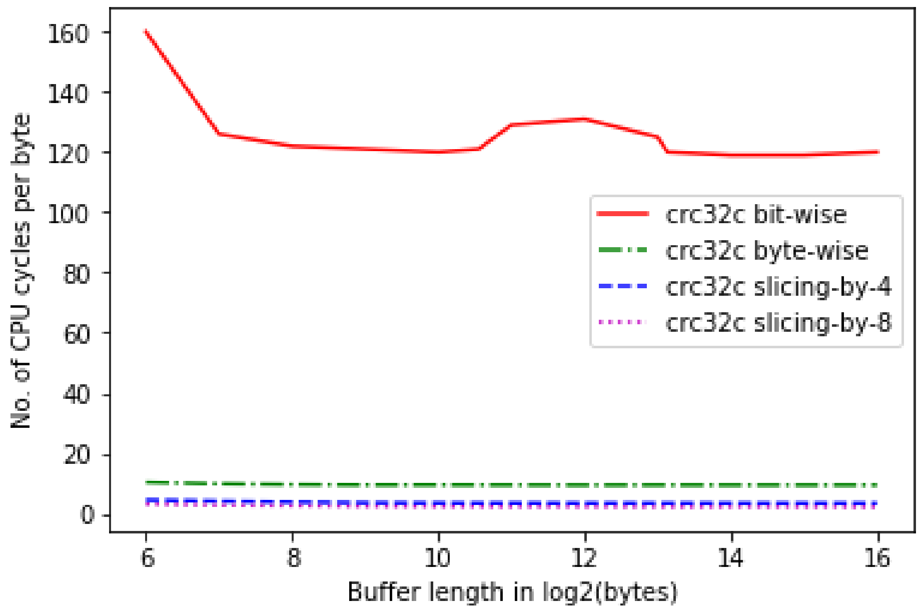


Figure 6.7: Performance comparison of naive bit-wise CRC32C vs table-driven CRC32C on Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

and PCLMULQDQ) and TCP one's complement checksum with 16-bit word. From the tables in A.5 and A.6, it can be seen that with larger buffer sizes (e.g., 1500 and 9000 bytes for both of the algorithms), there is little variation in the number of CPU cycles/byte when we added different number of extra bytes to those buffer sizes.

6.2 Benchmark study of checksum/ CRC algorithms on ARM

We have repeated the performance comparisons of different checksums and CRC algorithms on an ARM system that supports the hardware CRC32C instruction (Raspberry Pi Model 3B, ARM Cortex-A53, quadcore ARMv8 CPU). In order to do the measurements on ARM, we used the aarch64 generic timer register cntvct_el0 [43]. This is a 64-bit virtual timer count register, and it holds the virtual count value which is incremented at a rate of 19.2 MHz for our particular system. Normally, it varies in the range 1-50 MHz from system to system [43]. The system counter clock frequency was read from cntfrq_el0 register (a 64-bit register), and was found to be 19200000 Hz (19.2 MHz). We used the following command to determine the system's CPU frequency to be

1200 MHz.

```
sudo cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq
```

Originally, the Raspberry Pi came with a 32-bit Raspbian OS, and on doing “cat /proc/cpuinfo” it showed the model name as ARMv7 Processor rev 4 (v7l). We could not detect the CRC32C instruction. Later, we installed a 64-bit Gentoo Linux Operating system (OS) on the Raspberry Pi, and used the gcc compiler flag “-march=armv8-a+crc” to enable the hardware CRC32C instruction (polynomial 0x1EDC6F41) which is supported on some ARMv8-A CPUs. This CRC32C instruction calculates the CRC32C of the buffer in a similar manner to Intel’s CRC32 instruction. The ARM CRC32C instruction uses the reflected bit order for both the input data bytes as well as for the final CRC32C value. This CRC32C instruction has 8-bit, 16-bit, 32-bit, and 64-bit variants. ARM has another instruction named CRC32 which uses the Ethernet CRC32 polynomial 0x04C11DB7, and we have not considered that instruction in our work.

Table 6.2 shows the comparative computational performance measurement done on the Raspberry Pi Model 3B with ARM Cortex-A53 (aarch64 architecture) running Linux for the buffer sizes 64, 1500, and 9000 bytes. The actual cycle values that we got by reading the cntvct_el0 register value before and after the checksum/ CRC computation were converted to cycles/byte using the CPU frequency. We have used GCC optimization flag O3 to get these cycles/byte values for different algorithms over different buffer sizes. A more complete table A.2 shows the number of CPU cycles used per data byte over different data buffer sizes for different CRC32C and checksum implementations.

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte
crc32c bit-wise	64	173.75
	1500	163.125
	9000	161.875
crc32c nibble-wise	64	38.687
	1500	33.375
	9000	33.312
crc32c byte-wise	64	20.875
	1500	18.937
	9000	18.812
crc32c slicing-by-4	64	18.125
	1500	8
	9000	7.937
crc32c slicing-by-8	64	10.5
	1500	5.744
	9000	5.537
crc32c (serial usage of crc32c)	64	3.537
	1500	1.187
	9000	1.087
Adler-32	64	8.187
	1500	4.087
	9000	3.787
Fletcher-32	64	9.875
	1500	6.5
	9000	6.375
one’s comp 16-bit word (in C)	64	4.256
	1500	1.05
	9000	0.831
one’s comp 32-bit word (in C)	64	5.506
	1500	0.981
	9000	0.856
one’s comp 64-bit word (in C)	64	4.112
	1500	1.487
	9000	1.369

Table 6.2: Comparative computational performance of different CRC32C and Checksum implementations on Raspberry Pi Model 3B (ARM Cortex-A53)

Cavalcanti [50] has compared the performance of the CRC32 algorithm (with hardware CRC32 instruction used in a linear manner 4 byte-by-4 byte, followed by byte-by-byte) to the C program implementation of the zlib slicing-by-4 CRC32 algorithm [51] on an ARMv8 SoC. He found that the hardware instruction based CRC32 is almost 6 times faster than the slicing-by-4 CRC32. We can see in Fig. 6.8 that the serial computation of CRC32C with hardware CRC32C instruction is approximately 7.4 times faster than the CRC32C slicing-by-4 algorithm, and 5 times faster than the CRC32C slicing-by-8 algorithm.

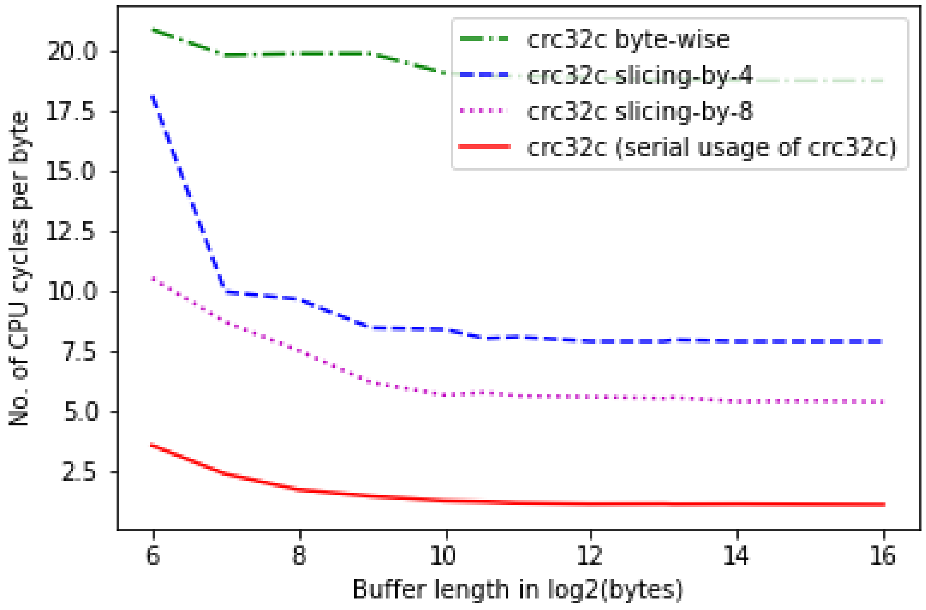


Figure 6.8: Performance comparison of table-driven CRC32C and hardware instruction based CRC32C on Raspberry Pi Model 3B (ARM Cortex-A53) with CPU max MHz: 1200 and CPU min MHz: 600

The CRC32C algorithm that we have tested on ARM computes the CRC32C linearly (8 byte-by-8 byte, followed by 4 byte-by-4 byte, and byte-by-byte). For ARM, we could not find an implementation similar to the one on Intel where the buffer is divided into 3 parts to speed up the CRC32C calculation. Fig. 6.9 shows the performance comparison of different checksum and CRC algorithms in terms of the number of CPU cycles/byte taken by these algorithms on the ARM

CPU. Serial computation of CRC32C on ARM using hardware CRC32C instruction is faster than the 16-bit word version of one's complement TCP checksum for buffer sizes up to 256 bytes. The hardware instruction based CRC32C algorithm is approximately 20% faster than the 16-bit TCP one's complement checksum at 64 bytes, and the 16-bit TCP one's complement checksum is roughly 13% faster than the hardware instruction based CRC32C at 1500 bytes. For buffer sizes greater than 256 bytes, the 16-bit word version of one's complement checksum is approximately 1.3 times faster than the serial computation of CRC32C. Nevertheless, the two algorithms offer similar performance. Consequently, using the CRC32C instruction on ARM systems with TCP will offer significantly enhanced error detection performance and similar computational performance compared to the TCP one's complement checksum.

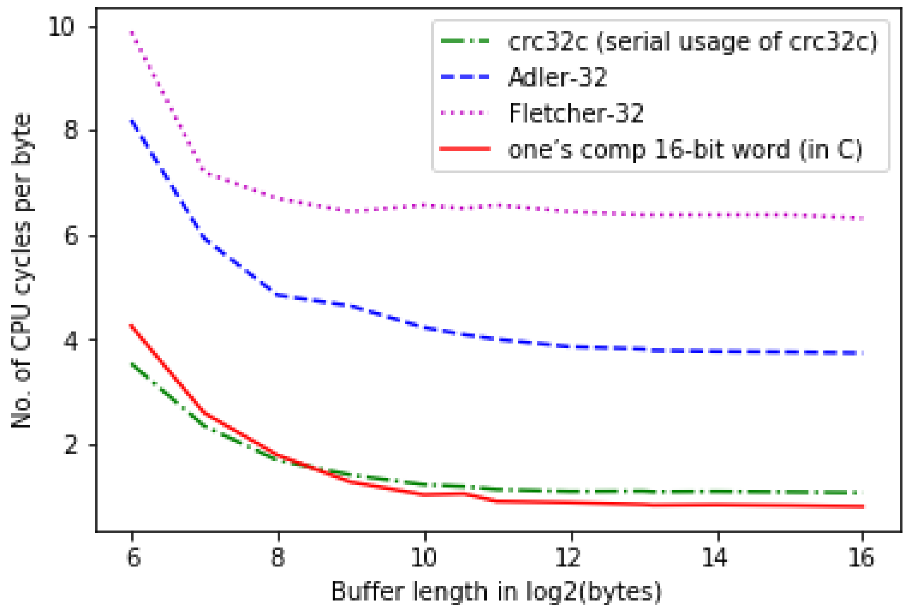


Figure 6.9: Cycles/byte performance of CRC32C using hardware CRC32C instruction, Adler-32 checksum, Fletcher-32 checksum, and TCP one's complement checksum on Raspberry Pi Model 3B (ARM Cortex-A53) with CPU max MHz: 1200 and CPU min MHz: 600

6.3 Benchmark study of checksum/ CRC algorithms on IBM POWER7

We also did the comparative computational performance evaluation on IBM PowerPC system. We used a POWER7 system with ppc64 architecture. This particular system has a model name "IBM,8246-L2T", and has Time Base of 512000000. The mftb instruction is used to read the Time Base (TB), which is a 64-bit register [52]. This register contains an unsigned integer of 64-bit, which increments at the rate of 512 MHz in our particular system. The clock frequency however showed 4228 MHz when we executed the command "cat /proc/cpuinfo". The mftb instruction is used before and after the checksum or CRC algorithm to know how many cycle counts (corresponding to the difference in the Time Base value) were used by the algorithm alone to execute [43]. The PowerPC architecture does not specify a relationship between the frequency at which the Time Base is updated and other frequencies, such as the CPU clock or bus clock in a PowerPC system [52]. The manual [52] also states that the Time Base update frequency is not required to be constant.

Table 6.3 shows the comparative computational performance measurement done on the IBM POWER7 (ppc64 architecture) running Linux for the buffer sizes 64, 1500, and 9000 bytes. The actual cycle values, that we got by reading the Time Base register value before and after the checksum/ CRC computation, were scaled up appropriately corresponding to the CPU frequency. We have used GCC optimization flag O3 to get these cycles/byte values for different algorithms over different buffer sizes. A more exhaustive table A.3 shows the number of CPU cycles used per data byte over different data buffer sizes for different CRC32C and checksum implementations.

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte
crc32c bit-wise	64	360.866
	1500	346.002
	9000	161.027
crc32c nibble-wise	64	17.754
	1500	15.855
	9000	15.689
crc32c byte-wise	64	11.396
	1500	10.074
	9000	9.909
crc32c slicing-by-4	64	7.977
	1500	5.879
	9000	5.764
crc32c slicing-by-8	64	5.904
	1500	3.295
	9000	3.096
Adler-32	64	3.113
	1500	2.106
	9000	1.99
Fletcher-32	64	2.882
	1500	2.023
	9000	1.949
one's comp 16-bit word (in C)	64	1.759
	1500	0.462
	9000	0.416
one's comp 32-bit word (in C)	64	1.594
	1500	0.745
	9000	0.706
one's comp 64-bit word (in C)	64	2.246
	1500	1.016
	9000	0.958

Table 6.3: Comparative computational performance of different CRC32C and Checksum implementations on IBM POWER7

The comparative computational performance of different checksum and CRC algorithms on an IBM POWER7 system running Linux kernel 3.10.0-957.12.2.el7.ppc64 can be seen in Fig. 6.10. Since we did not have access to a POWER8 system which supports the vector polynomial multiply sum (vpmsum) instruction for accelerating the CRC32 calculation; we just chose slicing-by-8 CRC32C algorithm, which is one of the best software based table-driven CRC32C algorithms, for the benchmark. For large buffer sizes, the 16-bit word version of the one's complement checksum algorithm is almost 7.4 times faster than the CRC32C slicing-by-8 on the POWER7 CPU, and 9.8 times faster than the CRC32C slicing-by-8 on Intel Xeon CPU.

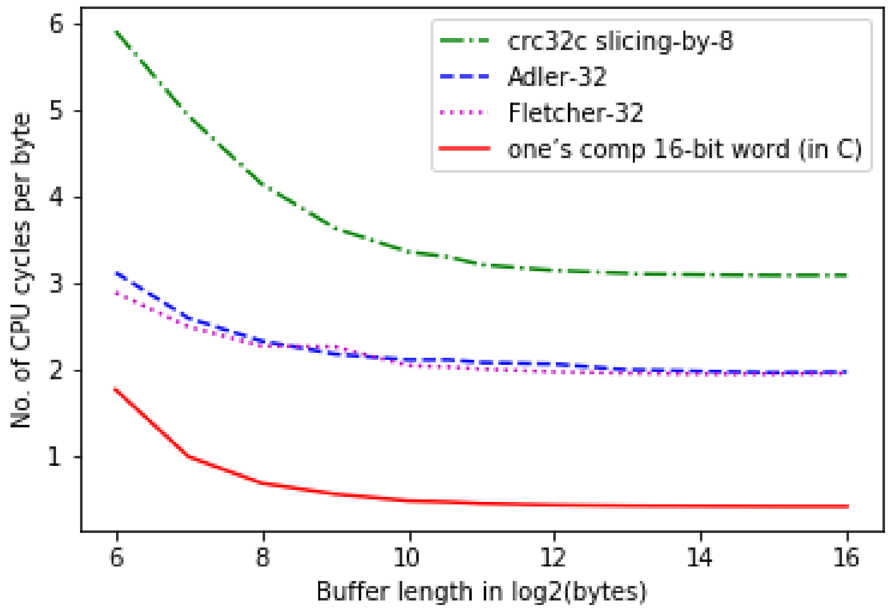


Figure 6.10: Cycles/byte performance of slicing-by-8 CRC32C, Adler-32 checksum, Fletcher-32 checksum, and TCP one's complement checksum on IBM POWER7 with timebase: 512000000 and clock: 4228MHz

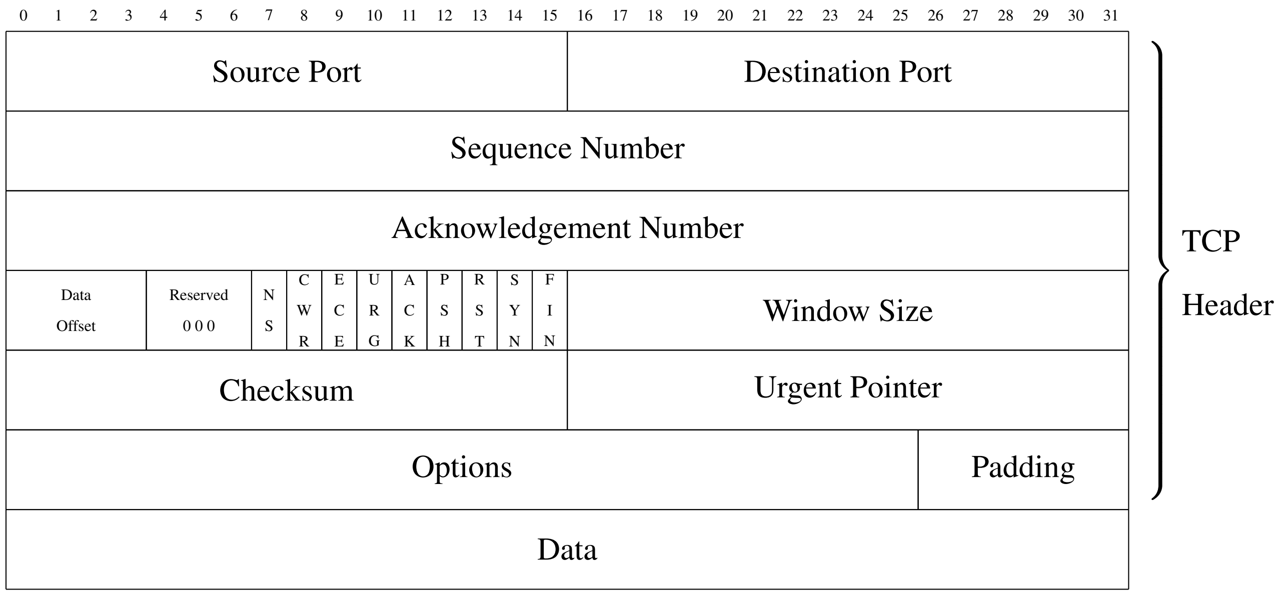
For the POWER8 architecture, Blanchard has compared the performance of the slicing-by-8 CRC32 algorithm to CRC32 using the vpmsum instruction over 32 KiB of input data and posted the results in his GitHub repository crc32-vpmsum [53]. He conducted the test on a 4.1 GHz POWER8 system and found that the CRC32 with vpmsum instruction takes up approximately

0.07 cycles/byte. He has also mentioned that the CRC32 computation, accelerated with vpmsum instruction, is about 41 times faster than the slicing-by-8 algorithm. This implies that the slicing-by-8 CRC32 takes approximately 3.058 cycles/byte on POWER8. We can see in figure 6.10 that slicing-by-8 CRC32C takes roughly 3.09 cycles/byte on POWER7 for large buffer sizes.

Since the POWER7 CPU that we have used in our research has different parameters as compared to the POWER8 CPU used by Blanchard (with unknown timebase information), we would not be able to directly compare his results with the results that we have found. However, we can see in Fig. 6.10 that for large buffer sizes (e.g., 9000 byte), the 16-bit word version of one's complement checksum algorithm is almost 7.4 times faster than the slicing-by-8 CRC32C algorithm, and Blanchard's results [53] suggest that the vpmsum accelerated CRC32 is almost 41 times faster than the slicing-by-8 CRC32C algorithm on POWER8. So, we can make an educated guess that the vpmsum accelerated CRC32 would computationally outperform the TCP one's complement checksum on POWER8.

7. CRC32C IN TCP THROUGH TCP OPTION

For an IP datagram, the TCP segment is the payload and it is encapsulated inside the IP datagram. The TCP segment consists of the TCP header and data. The TCP segment looks as follows [1] [54]:



The Data Offset field is a 4-bit entity. This field gives the number of 32 bit (4 byte) words present in the TCP header [1]. It indicates where the data portion starts with respect to the start of the TCP header. The TCP header, including all the TCP Options, must always be an integral multiple of 32 bits, and the TCP Option field is limited to a maximum of 40 bytes.

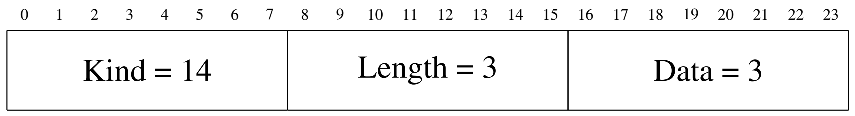
Once the mandatory 20 byte TCP header ends, the TCP Options begin from there. In terms of length, the TCP Options are multiples of 8 bits. The TCP Option can be of two types [1]: one with a single byte of option-kind, and the other variant has a single byte of option-kind followed by a single byte of option-length, and then followed by one or more bytes of the actual option-data. The option-length field contains the sum total of the size (in bytes) of option-kind, option-length, and the option-data fields.

Originally, TCP had only three TCP Options: End of option list, No-Operation, and Maximum

Segment Size [1]. TCP Window Scale, Selective Acknowledgments (SACK) and Timestamps Options were added later as extensions to TCP in order to further improve performance. Here, we propose the use of CRC32C to replace the TCP one's complement checksum using a previously proposed TCP Option.

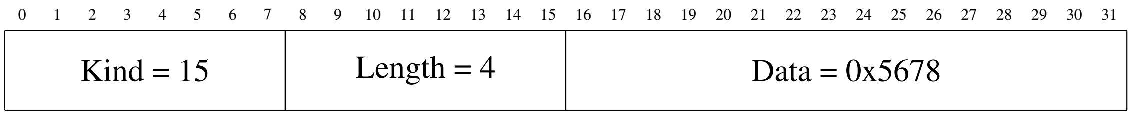
RFC 1146 [19] proposed the use of a TCP Alternate Checksum Option in the early 1990s. However, in the year 2011, RFC 6247 [20] reclassified RFC 1146 to historic status since RFC 1146 had not seen any widespread deployment. We can potentially revive RFC 1146 and include the CRC32C as one of the alternate checksum algorithms. RFC 1146 had defined three different option-data values for three different checksums as follows: 0 for TCP Checksum, 1 for an 8-bit Fletcher's algorithm, and 2 for a 16-bit Fletcher's algorithm.

Since there is already a provision for a TCP Alternate Checksum Request Option and the TCP Alternate Checksum Data Option in RFC 1146 [19], we propose to suitably tailor these options in order to retrofit the existing (though now obsolete) TCP Options with CRC32C. As 0, 1 and 2 had already been defined as option-data values in the TCP Alternate Checksum Request Option, we can use the next available value, which is 3, as the option-data value corresponding to the CRC32C algorithm. The TCP Alternate Checksum Request Option, with the proposed CRC32C algorithm, looks as follows:



We propose to use the option-data field in the TCP Alternate Checksum Data Option to store the two least significant bytes of the calculated CRC32C value, and the Checksum field in the TCP header to carry the two most significant bytes of the CRC32C value. The option-length field in the TCP Alternate Checksum Data Option should always contain 4, as the option-data field in the TCP Alternate Checksum Data Option will always carry the two LSBs of the calculated CRC32C value, and the option-kind and option-length fields are 1 byte each. For example, if the calculated

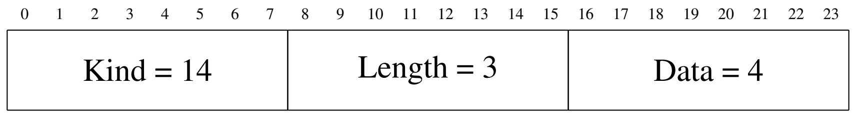
CRC32C turns out to be 0x12345678, then the TCP Alternate Checksum Data Option with the proposed CRC32C algorithm looks as follows:



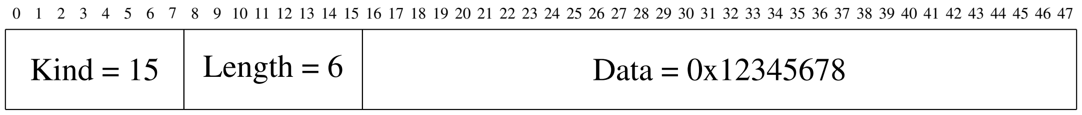
In an Internet-Draft (now expired), Biswas [55] has proposed the use of CRC32C in TCP using a TCP Option. However, he has explicitly recommended the use of CRC32C for Jumbo frames only, and has mentioned that the Ethernet CRC is sufficient to protect the Ethernet MTU sized frames. Also, he has proposed to eliminate the pseudo header field while calculating the CRC32C of a TCP segment, and the reason being the pseudo header content gets checked three times already (in Ethernet CRC, IP header checksum, and TCP checksum). He has mentioned two approaches to negotiate the use of CRC32C in TCP. In the first approach, he has proposed to put the entire 4 bytes of the calculated CRC32C in the TCP option-data field, and in the second approach, he has proposed to use the TCP Alternate Checksum Option (as described in RFC 1146). However, we chose to go with his second approach; as with his first approach (uses 6 bytes in total for TCP Option), we would be able to do only 2 SACK blocks (18 bytes) along with Time Stamp Option (10 bytes), and CRC32C Option (6 bytes). With his second approach, we can potentially do 3 SACK blocks (26 bytes), Time Stamp Option (10 bytes), and TCP Alternate Checksum Data Option (4 bytes) altogether utilizing all the 40 bytes of the TCP Option field. We propose to use CRC32C for TCP segments (carrying data and/ or ACK) of all sizes and to also include the pseudo header in the TCP CRC32C calculation [56].

To calculate the CRC32C, we need to consider the 12 byte TCP pseudo header (with the same meaning from the original TCP checksum computation) along with the TCP segment. Biswas [55] has assumed that for the CRC32C computation to proceed, all the data (over which the CRC32C would be computed) needed to be present in a contiguous manner. However, the CRC32C can be potentially calculated incrementally. So, with an initial CRC value of 0xFFFFFFFF, first the CRC32C of the TCP pseudo header can be computed. The resulting CRC value can be used as initial CRC value to calculate the CRC32C of the TCP header followed by data. The Checksum field in the TCP header, and the option-data field in the TCP Alternate Checksum Data Option are set to 0 before we start computing the CRC32C of the TCP segment. Once we calculate the CRC32C, we store the CRC32C value as described earlier.

Things become complicated with Network Address Translation (NAT), which is a common practice these days in the small office/home office environment and in the enterprise environment, e.g., Texas A&M University uses NAT for Wi-Fi connections. Hence, we need to reconsider the use of TCP Option to calculate the CRC32C of a TCP segment behind a NAT device. In the basic NAT model, the source IP address of the outgoing packets (originated behind the NAT device) needs to be changed from the private IP address of the initiator host (behind the NAT device) to the routable IP address of the NAT device. Similarly, for the incoming packets, the destination IP address needs to be changed from the routable IP address of the NAT device to the private IP address of the actual destination host [57]. Things become even more complicated for the Network Address Port Translation (NAPT) model, where the source TCP/UDP port may also need to be changed for the outbound packets, and the destination TCP/UDP port may need to be changed for the inbound packets along with the inevitable source and destination IP address translations [57]. Because of these IP address/port translations, the TCP checksum is recomputed (modified incrementally) in the NAT devices, which is why the NAT devices violate the end-to-end principle. If we use CRC32C instead of one's complement checksum in TCP, then the CRC32C needs to be recomputed in the NAT devices from the start of the new pseudo header (after the address translation), followed by the modified TCP header (because of possible port translation) and data, and this would take more time when compared to the TCP checksum incremental modification. So, with NAT, we propose to calculate the CRC32C of the TCP data only along with calculating the standard TCP checksum over the entire TCP segment (including the pseudo header) in the end hosts. The NAT device now just needs to recompute the TCP checksum, and it need not calculate the CRC32C again (as the data has not changed). The destination host upon receiving the TCP data can compute the CRC32C on it so as to verify with the received CRC32C value which was calculated just on the TCP data. A TCP segment would be discarded if either or both the CRC32C and one's complement checksum were in error. Previously, we have used "3" as the option-data value in the TCP Alternate Checksum Request Option corresponding to the CRC32C algorithm that calculates the CRC32C over the TCP pseudo header, followed by the TCP header and data. For hosts behind a NAT device, we propose to use the next available value, which is 4, as the option-data value in the TCP Alternate Checksum Request Option corresponding to the CRC32C algorithm that calculates the CRC32C of the TCP data only. With NAT, the TCP Alternate Checksum Request Option looks as follows:

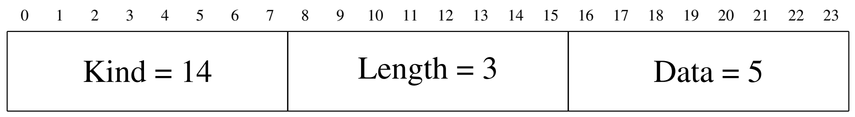


For hosts behind a NAT device, we propose to use the option-data field in the TCP Alternate Checksum Data Option to store the entire four bytes of the calculated CRC32C value, and the Checksum field in the TCP header will carry the regular TCP checksum value. The option-length field in the TCP Alternate Checksum Data Option now should always be 6, as the size of option-data field in the TCP Alternate Checksum Data Option is always 4, and the option-kind and option-length fields are 1 byte each. For example, if the calculated CRC32C turns out to be 0x12345678, then the TCP Alternate Checksum Data Option with the proposed CRC32C algorithm to be used with NAT looks as follows:

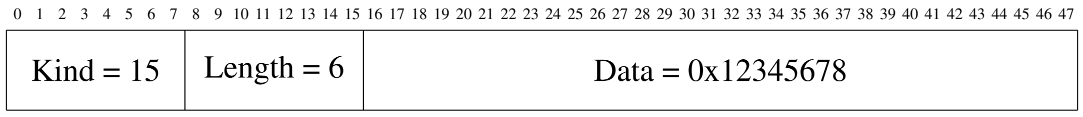


TCP checksum offloading is commonly done these days even though the practice differs from vendor to vendor [58] [59]. With checksum offload, we are basically relying on the network interface card (NIC) hardware to perform the checksum computation, and it therefore saves CPU time.

The fast hardware instruction based implementations of the CRC32C on different platforms can be used to compute the CRC32C of a TCP segment (including the pseudo header) in the normal fashion, and we can put those 4 bytes of CRC32C inside a TCP Option. The standard TCP checksum value (calculated via checksum offloading) will be stored in the Checksum field of the TCP header. We need to define a new option-data value in the TCP Alternate Checksum Request Option corresponding to the CRC32C algorithm that calculates the CRC32C of the entire TCP segment including the pseudo header. We can use the next available value, which is 5, as the option-data value in the TCP Alternate Checksum Request Option for the CRC32C algorithm that calculates the CRC32C of the entire TCP segment including the pseudo header. So, with TCP checksum offload, the TCP Alternate Checksum Request Option looks as follows:



For hosts doing TCP checksum offload, we propose to use the option-data field in the TCP Alternate Checksum Data Option to store the entire four bytes of the calculated CRC32C value, and the Checksum field in the TCP header will carry the regular TCP checksum value which is calculated via checksum offload. The option-length field in the TCP Alternate Checksum Data Option now should always be 6, as the size of option-data field in the TCP Alternate Checksum Data Option is always 4, and the option-kind and option-length fields are 1 byte each. For example, if the calculated CRC32C turns out to be 0x12345678, then the TCP Alternate Checksum Data Option with the proposed CRC32C algorithm to be used with TCP checksum offload looks as follows:



There can be different combinations of the host with respect to TCP checksum offload and host behind a NAT device. The following table summarizes which option-data value to choose in the TCP Alternate Checksum Request Option corresponding to different scenarios. The host should choose the appropriate TCP Alternate Checksum Data Option corresponding to the chosen TCP Alternate Checksum Request Option.

Host Scenario		Option-data value in the TCP Alternate Checksum Request Option
TCP checksum offload enabled ?	Host behind a NAT device ?	
No	No	3
No	Yes	4
Yes	No	5
Yes	Yes	4

Table 7.1: Host TCP checksum offload and NAT condition check to use appropriate option-data value in the TCP Alternate Checksum Request Option

Based on the types of host scenarios presented in Table 7.1, there can be nine possible combinations of option-data values in the TCP Alternate Checksum Request Option for the sender and receiver. For all the different combinations, the final option-data value that the sender chooses is shown in Table 7.2. For example, if the sender opts for 3 as the option-data value in the TCP Alternate Checksum Request Option and the receiver responds with the option-data value 4, then the sender should select 4 as the final option-data value in the TCP Alternate Checksum Request Option and should use the corresponding CRC32C algorithm in the TCP Alternate Checksum Data Option from the first non-SYN segment onwards.

The actual negotiation mechanism to use the CRC32C is identical to the method described in RFC 1146. At the time of connection establishment, the TCP Alternate Checksum Request Option, with option-data value equal to 3, can be sent by the sender in the SYN segment. If the acknowledging receiver also sends the TCP Alternate Checksum Request Option, with option-data value equal to 3 in the SYN-ACK segment, then the CRC32C (calculated over the entire TCP

Option-data value in TCP Alternate Checksum Request Option sent by:		Final option-data value chosen
Sender	Receiver	
3	3	3
3	4	4
3	5	5
4	3	4
4	4	4
4	5	4
5	3	5
5	4	4
5	5	5

Table 7.2: Final option-data value chosen based on the option-data values sent by the sender and receiver in the TCP Alternate Checksum Request Option

segment including the pseudo header) would be used from this point onwards during the data transfer phase via the TCP Alternate Checksum Data Option. The TCP segment with SYN or RST flag set must use the regular TCP checksum as prescribed in RFC 1146. As per RFC 1122, any TCP implementation not recognizing these TCP Options should silently ignore it. Consequently, an end station proposing CRC32C must receive a TCP Alternate Checksum Request Option in response, or it must use the standard TCP checksum. Although we have implicitly assumed IPv4 everywhere, the CRC32C can be used over IPv6 as well. We just need to use the TCP pseudo header corresponding to IPv6 [60], in order to calculate the CRC32C value as described earlier.

Presumably, the end hosts will check if the hardware CRC32C or a carry-less mod2 multiply instruction is available with them or not, if the TCP checksum offload is enabled or not, and if they are behind a NAT device or not. Then accordingly they should choose the right option-data value in the TCP Alternate Checksum Request Option in the SYN segment during the connection establishment phase.

In the future, if hardware instructions are available for even better CRC32 polynomials (e.g., the Koopman polynomial 0xBA0DC66B that offers HD=6 at MTU=1500 bytes), then the CRC32 with those polynomials can be added to the list of alternate checksum algorithms, and can be used in a similar fashion to calculate the CRC32 of a TCP segment.

8. SUMMARY AND CONCLUSIONS

We presented benchmarks of many variants of addition-based checksums and CRC algorithms in this thesis. We developed a benchmark routine that tested the implementations of all these algorithms for their correctness. We also calculated the number of CPU cycles per byte for each of these algorithms in order to compare them based on their computational performance.

Today, all the Intel CPUs have hardware CRC32 and PCLMULQDQ (carry less multiplication) instructions which potentially can be used to efficiently calculate the CRC32C of a TCP segment. A growing number of modern ARM processors also have the hardware CRC32C instruction. As the CRC32C algorithm has found its utility in some of the major applications (iSCSI, SCTP, Btrfs, ext4), we are expecting that all the ARM processors will have the hardware CRC32C instruction in the next few years. The ARMv8.1-A architecture specification from 2014 made the CRC instruction mandatory. Other architectures (e.g., POWER8, IBM Z Series, SPARC) have also facilitated CRC32 hardware acceleration with a carry less multiplication instruction.

The one's complement checksum is still in use today in TCP, some 40 years after it was originally incorporated into IP, TCP, and UDP. While there have been papers and RFCs proposing stronger TCP error detection algorithms and ways to accomplish this using a TCP Option, none of these efforts have led to a change in the TCP checksum. The advent of hardware CRC instructions in most of the standard computer architectures, which provide better performance than the existing TCP checksum algorithm, offers the opportunity to reconsider using a TCP Option for CRC32C-based error detection. Our results show that on Intel Core i3-2330M, the CRC32C hardware instruction implementation is approximately 38% faster than the 16-bit TCP one's complement checksum at 1500 bytes and the 16-bit TCP one's complement checksum is roughly 11% faster than the hardware instruction based CRC32C at 64 bytes. On ARM Cortex-A53, the hardware CRC32C algorithm is approximately 20% faster than the 16-bit TCP one's complement checksum at 64 bytes, and the 16-bit TCP one's complement checksum is roughly 13% faster than the hardware instruction based CRC32C at 1500 bytes. Given that, many applications work on top of TCP, and they depend on TCP to provide data integrity; it is time to reconsider a TCP Option, as discussed in this research, to use the hardware instruction based CRC32C in TCP in order to achieve much better end-to-end error detection performance.

REFERENCES

[1] J. Postel, "Transmission Control Protocol," RFC 793, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[2] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," *SIGCOMM Comput. Commun. Rev.*, vol. 30, pp. 309–319, Aug. 2000.

[3] "Hardware Only (HO) features and technologies | Microsoft Docs." `https://docs.microsoft.com/en-us/windows-server/networking/technologies/hpn/hpn-hardware-only-features`, November 2018.

[4] E. Jones, "How both TCP and Ethernet checksums fail." `https://www.evanjones.ca/tcp-and-ethernet-checksums-fail.html`, October 2015.

[5] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna, "Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations," Tech. Rep. RFC3385, RFC Editor, Sept. 2002.

[6] J. Stone, R. Stewart, and D. Otis, "Stream Control Transmission Protocol (SCTP) Checksum Change," Tech. Rep. RFC3309, RFC Editor, Sept. 2002.

[7] *Intel® SSE4 Programming Reference*, July 2007.

[8] "White Paper: Intel® Next Generation Microarchitecture (Nehalem)." `https://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf`.

[9] "ARM Information Center." `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0801g/awi1476352818103.html`.

[10] G. Castagnoli, S. Brauer, and M. Herrmann, "Optimization of cyclic redundancy-check codes with 24 and 32 parity bits," *IEEE Transactions on Communications*, vol. 41, pp. 883–892, June 1993.

[11] T. Maxino and P. Koopman, "The Effectiveness of Checksums for Embedded Control Networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, pp. 59–72, Jan. 2009.

[12] "IEEE standard for ethernet," *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, March 2016.

[13] M. Bhatia, "Catching Corrupted OSPF Packets!." `https://routingfreak.wordpress.com/2011/03/01/catching-corrupted-ospf-packets/`, March 2011.

[14] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, pp. 277–288, Nov. 1984.

[15] R. M. Gates, *From the Shadows: The Ultimate Insider's Story of Five Presidents and How They Won the Cold War*. Simon & Schuster, 2007.

[16] "Causes of False Missile Alerts: The Sun, the Moon and a 46-Cent Chip." `https://www.nytimes.com/2018/01/13/us/false-alarm-missile-alerts.html`.

[17] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier, "TCP performance re-visited," in *2003 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS 2003.*, pp. 70–79, March 2003.

[18] R. Braden, D. Borman, C. Partridge, and W. W. Plummer, "Computing the Internet checksum," RFC 1071, RFC Editor, September 1988. `http://www.rfc-editor.org/rfc/rfc1071.txt`.

[19] J. Zweig and C. Partridge, "TCP alternate checksum options," RFC 1146, RFC Editor, March 1990. `http://www.rfc-editor.org/rfc/rfc1146.txt`.

[20] L. Eggert, "Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status," RFC 6247, RFC Editor, May 2011. `http://www.rfc-editor.org/rfc/rfc6247.txt`.

[21] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "Stream Control Transmission Protocol," RFC 2960, RFC Editor, October 2000. `http://www.rfc-editor.org/rfc/rfc2960.txt`.

[22] L. P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," RFC 1950, RFC Editor, May 1996. `http://www.rfc-editor.org/rfc/rfc1950.txt`.

[23] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and CRCs over real data," *IEEE/ACM Transactions on Networking*, vol. 6, pp. 529–543, Oct 1998.

[24] P. Koopman, "32-bit cyclic redundancy codes for Internet applications," in *Proceedings International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 459–468, IEEE Comput. Soc, 2002.

[25] J. Daugherty, "Understanding iSCSI Digests:."

[26] M. E. Kounavis and F. L. Berry, "Novel Table Lookup-Based Algorithms for High-Performance CRC Generation," *IEEE Transactions on Computers*, vol. 57, pp. 1550–1560, Nov. 2008.

[27] D. V. Sarwate, "Computation of cyclic redundancy checks via table look-up," *Commun. ACM*, vol. 31, pp. 1008–1013, Aug. 1988.

[28] S. Gueron, "Speeding up CRC32c computations with Intel CRC32 instruction," *Information Processing Letters*, vol. 112, pp. 179–185, Feb. 2012.

[29] L. L. Peterson and B. S. Davie, *Computer Networks: a systems approach*, pp. 97–102. Morgan Kaufmann, 2012.

[30] J. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Transactions on Communications*, vol. 30, pp. 247–252, January 1982.

[31] A. Nakassis, "Fletcher's error detection algorithm: How to implement it efficiently and how to avoid the most common pitfalls," *SIGCOMM Comput. Commun. Rev.*, vol. 18, pp. 63–88, Oct. 1988.

[32] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, pp. 228–235, Jan 1961.

[33] R. Williams, "A Painless Guide to CRC Error Detection Algorithms." `http://www.ross.net/crc/download/crc_v3.txt`, August 1993.

[34] T. V. Ramabadran and S. S. Gaitonde, "A tutorial on CRC computations," *IEEE Micro*, vol. 8, pp. 62–75, Aug 1988.

[35] M. Barr, "CRC Series, Part 3: CRC Implementation Code in C/C++." `https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code`, January 2000.

[36] "Intrinsics | Intel® C++ Compiler 19.0." `https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intrinsics`, April 2019.

[37] "Fast CRC Computation for iSCSI Polynomial Using CRC32 Instruction." `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/crc-iscsi-polynomial-crc32-instruction-paper.pdf`, April 2011.

[38] M. Adler, "Implementing SSE 4.2's CRC32C in software." `https://stackoverflow.com/questions/17645167/implementing-sse-4-2s-crc32c-in-software`, July 2013.

[39] S. Reifegerste, "CRC Calculation." `http://www.zorc.breitbandkatze.de/crc.html`.

[40] S. Brumme, "Fast CRC32." `https://create.stephan-brumme.com/crc32/`, November 2011.

[41] B. Even, "crcbench." `https://github.com/baruch/crcbench/`, April 2014.

[42] "Intel CRC Benchmark Application." https://github.com/intel/soft-crc, January 2017.

[43] "Benchmark." https://github.com/google/benchmark.

[44] F. Toth, "Highly optimized CRC32C lib and benchmark." https://github.com/htot/crc32c, November 2017.

[45] B. Molkenthin, "Sunshine's Homepage - Online CRC Calculator Javascript." http://www.sunshine2k.de/coding/javascript/crc/crc_js.html, May 2015.

[46] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D*, 2016.

[47] "Choosing a CRC polynomial and associated method for Fast CRC Computation on Intel® Processors." https://pdfs.semanticscholar.org/b01a/0f242ce5537d806feefeff17cf72cc257946.pdf, August 2012.

[48] "The TCP/IP Checksum." https://locklessinc.com/articles/tcp_checksum/.

[49] D. P. Garcia, "Fast checksum computation." https://blogs.igalia.com/dpino/2018/06/14/fast-checksum-computation/, June 2018.

[50] A. Cavalcanti, "Using ARMv8 CRC32 specific instruction." https://chromium.googlesource.com/chromium/src.git/+/35988c821c051a57e30c76f9fcd87b7b677bd9bd, November 2017.

[51] M. Adler, "zlib CRC32." https://github.com/madler/zlib/blob/master/crc32.c, January 2017.

[52] *PowerPC Operating Environment Architecture, Book III, Version 2.02*, January 2005.

[53] A. Blanchard, "Accelerated CRC32 for POWER8 using vpmsum instructions." https://github.com/antonblanchard/crc32-vpmsum, February 2015.

[54] N. Spring, D. Wetherall, and D. Ely, “Robust Explicit Congestion Notification (ECN) Signaling with Nonces,” RFC 3540, RFC Editor, June 2003.

[55] A. Biswas, “Support for Stronger Error Detection Codes in TCP for Jumbo Frames.” https://tools.ietf.org/html/draft-ietf-tcpm-anumita-tcp-stronger-checksum-00, May 2010.

[56] D. P. Reed, “Purpose of pseudo header in TCP checksum.” http://www.postel.org/pipermail/end2end-interest/2005-February/004616.html, February 2005.

[57] P. Srisuresh and K. Egevang, “Traditional ip network address translator (traditional nat),” RFC 3022, RFC Editor, January 2001.

[58] J. Chase, “Checksum Offloading.” https://www2.cs.duke.edu/ari/trapeze/freenix/node7.html, August 1999.

[59] “TCP checksum offload.” https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.performance/tcp_checksum_offload.htm.

[60] S. E. Deering and R. M. Hinden, “Internet Protocol, Version 6 (IPv6) Specification,” RFC 2460, RFC Editor, December 1998. http://www.rfc-editor.org/rfc/rfc2460.txt.

APPENDIX A

COMPARATIVE PERFORMANCE EVALUATION OF DIFFERENT CRC/ CHECKSUM ALGORITHMS

Table A.1: Comparative computational performance of different CRC32C and Checksum implementations on Intel CPUs over different buffer sizes (with GCC optimization flag O3)

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:	
		Intel Core i3-2330M	Intel Xeon
crc32c bit-wise	64	160	109
	128	126	110
	256	122	103
	512	121	102
	1024	120	101
	1500	121	101
	2048	129	101
	4096	131	101
	8192	125	101
	9000	120	101
	16384	119	101
	32768	119	101
	65536	120	100
crc32c nibble-wise	64	17.8	14.9
	128	17.5	14.4
	256	17.4	14.3
	512	17.6	14.1
	1024	17.4	14.1
	1500	17.3	14.3
	2048	17.3	14
	4096	17.3	14.1
	8192	17.2	14
	9000	17.3	14
	16384	17.3	14
	32768	17.3	14
	65536	17.3	14
	64	10.4	7.8
	128	9.87	7.39
	256	9.61	7.18

Table A.1: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:	
		Intel Core i3-2330M	Intel Xeon
crc32c byte-wise	512	9.47	7.11
	1024	9.48	7.05
	1500	9.45	7.03
	2048	9.44	7.1
	4096	9.4	7.02
	8192	9.43	7.01
	9000	9.43	7.01
	16384	9.41	7.02
	32768	9.41	7.01
	65536	9.42	7
crc32c slicing-by-4	64	4.5	3.53
	128	4.12	3.07
	256	3.84	2.81
	512	3.62	2.67
	1024	3.55	2.61
	1500	3.51	2.61
	2048	3.51	2.58
	4096	3.48	2.56
	8192	3.47	2.56
	9000	3.47	2.56
	16384	3.47	2.55
	32768	3.47	2.54
	65536	3.46	2.54
crc32c slicing-by-8	64	3.14	2.66
	128	2.74	2.19
	256	2.57	1.94
	512	2.34	1.88
	1024	2.3	1.72
	1500	2.27	1.72
	2048	2.23	1.68
	4096	2.22	1.66
	8192	2.2	1.66
	9000	2.21	1.65
	16384	2.2	1.65
	32768	2.19	1.64
	65536	2.18	1.64
	64	1.23	1.04
	128	1.01	0.734

Table A.1: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:	
		Intel Core i3-2330M	Intel Xeon
crc32c Intel Intrinsic	256	0.702	0.584
	512	0.601	0.507
	1024	0.554	0.469
	1500	0.538	0.462
	2048	0.543	0.461
	4096	0.514	0.444
	8192	0.508	0.442
	9000	0.509	0.437
	16384	0.504	0.435
	32768	0.504	0.434
	65536	0.504	0.433
crc32c (crc32 + pclmulqdq based recombination)	64	1.25	1.23
	128	0.868	0.714
	256	0.49	0.401
	512	0.333	0.272
	1024	0.255	0.194
	1500	0.211	0.168
	2048	0.191	0.152
	4096	0.164	0.138
	8192	0.156	0.13
	9000	0.151	0.128
	16384	0.146	0.126
	32768	0.147	0.122
	65536	0.146	0.121
crc32c (crc32 + table-based recombination)	64	1.15	1.03
	128	0.753	0.673
	256	0.569	0.5
	512	0.475	0.413
	1024	0.355	0.256
	1500	0.37	0.279
	2048	0.315	0.222
	4096	0.23	0.175
	8192	0.208	0.169
	9000	0.202	0.17
	16384	0.184	0.16
	32768	0.156	0.134
	65536	0.153	0.129
	64	2.76	2.37

Table A.1: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte: Intel Core i3-2330M	Intel Xeon
Adler-32	128	1.96	1.69
	256	1.6	1.37
	512	1.43	1.19
	1024	1.33	1.11
	1500	1.31	1.1
	2048	1.29	1.07
	4096	1.29	1.04
	8192	1.28	1.06
	9000	1.29	1.06
	16384	1.3	1.05
	32768	1.3	1.05
	65536	1.31	1.05
Fletcher-32	64	1.95	1.52
	128	1.72	1.33
	256	1.39	1.1
	512	1.28	1.04
	1024	1.17	0.997
	1500	1.18	0.972
	2048	1.16	0.965
	4096	1.13	0.949
	8192	1.12	0.943
	9000	1.12	0.944
	16384	1.11	0.935
	32768	1.11	0.934
	65536	1.11	0.933
one’s comp 16-bit word (in C)	64	1.29	1.04
	128	0.801	0.623
	256	0.533	0.408
	512	0.379	0.287
	1024	0.31	0.248
	1500	0.275	0.227
	2048	0.251	0.206
	4096	0.225	0.186
	8192	0.208	0.176
	9000	0.207	0.174
	16384	0.202	0.171
	32768	0.198	0.168

Table A.1: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:	
		Intel Core i3-2330M	Intel Xeon
	65536	0.199	0.167
one's comp 32-bit word (in C)	64	1.38	1.04
	128	0.845	0.681
	256	0.61	0.458
	512	0.428	0.309
	1024	0.331	0.26
	1500	0.285	0.23
	2048	0.266	0.211
	4096	0.232	0.187
	8192	0.209	0.178
	9000	0.214	0.174
	16384	0.204	0.17
	32768	0.2	0.168
	65536	0.2	0.166
one's comp 64-bit word (in x86_64 assembly)	64	1.13	0.964
	128	0.696	0.644
	256	0.483	0.406
	512	0.361	0.306
	1024	0.31	0.261
	1500	0.292	0.248
	2048	0.284	0.24
	4096	0.269	0.229
	8192	0.262	0.223
	9000	0.259	0.222
	16384	0.256	0.22
	32768	0.254	0.218
	65536	0.253	0.218

Table A.2: Comparative performance evaluation of different CRC32C and Checksum implementations on Raspberry Pi Model 3B (ARM Cortex-A53) over different buffer sizes (with GCC optimization flag O3)

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
crc32c bit-wise	64	173.75
	128	169.375
	256	166.875
	512	165
	1024	162.5
	1500	163.125
	2048	162.5
	4096	162.5
	8192	161.875
	9000	161.875
	16384	161.875
	32768	161.25
	65536	161.25
crc32c nibble-wise	64	38.687
	128	37.937
	256	33.437
	512	33.875
	1024	33.375
	1500	33.375
	2048	33.5
	4096	33.5
	8192	33.187
	9000	33.312
	16384	33.25
	32768	33.25
	65536	33.187
crc32c byte-wise	64	20.875
	128	19.812
	256	19.875
	512	19.875
	1024	19.062
	1500	18.937
	2048	18.937
	4096	18.875
	8192	18.75
	9000	18.812

Table A.2: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
	16384	18.75
	32768	18.75
	65536	18.75
crc32c slicing-by-4	64	18.125
	128	9.937
	256	9.625
	512	8.437
	1024	8.375
	1500	8
	2048	8.062
	4096	7.875
	8192	7.875
	9000	7.937
	16384	7.875
	32768	7.875
	65536	7.875
crc32c slicing-by-8	64	10.5
	128	8.687
	256	7.5
	512	6.156
	1024	5.637
	1500	5.744
	2048	5.606
	4096	5.562
	8192	5.5
	9000	5.537
	16384	5.369
	32768	5.394
	65536	5.356
crc32c (serial usage of crc32c)	64	3.537
	128	2.344
	256	1.687
	512	1.412
	1024	1.225
	1500	1.187
	2048	1.131
	4096	1.094
	8192	1.1

Table A.2: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
	9000	1.087
	16384	1.094
	32768	1.081
	65536	1.069
Adler-32	64	8.187
	128	5.925
	256	4.844
	512	4.637
	1024	4.225
	1500	4.087
	2048	4
	4096	3.856
	8192	3.812
	9000	3.787
	16384	3.769
	32768	3.756
	65536	3.737
Fletcher-32	64	9.875
	128	7.187
	256	6.687
	512	6.437
	1024	6.562
	1500	6.5
	2048	6.562
	4096	6.437
	8192	6.375
	9000	6.375
	16384	6.375
	32768	6.375
	65536	6.312
one's comp 16-bit word (in C)	64	4.256
	128	2.594
	256	1.787
	512	1.275
	1024	1.031
	1500	1.05
	2048	0.906
	4096	0.881

Table A.2: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
	8192	0.844
	9000	0.831
	16384	0.837
	32768	0.825
	65536	0.806
one's comp 32-bit word (in C)	64	5.506
	128	3.181
	256	2.069
	512	1.394
	1024	1.125
	1500	0.981
	2048	0.956
	4096	0.856
	8192	0.862
	9000	0.856
	16384	0.831
	32768	0.831
	65536	0.812
one's comp 64-bit word (in C)	64	4.112
	128	3.025
	256	2.137
	512	1.744
	1024	1.531
	1500	1.487
	2048	1.456
	4096	1.344
	8192	1.362
	9000	1.369
	16384	1.356
	32768	1.337
	65536	1.319

Table A.3: Comparative performance evaluation of different CRC32C and Checksum implementations on IBM POWER7 over different buffer sizes (with GCC optimization flag O3)

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
crc32c bit-wise	64	360.866
	128	353.434
	256	354.26
	512	347.653
	1024	347.653
	1500	346.002
	2048	346.002
	4096	345.176
	8192	344.35
	9000	161.027
	16384	161.027
	32768	161.027
	65536	161.027
crc32c nibble-wise	64	17.754
	128	17.176
	256	16.598
	512	16.268
	1024	16.02
	1500	15.855
	2048	16.103
	4096	15.855
	8192	15.772
	9000	15.69
	16384	15.69
	32768	15.607
	65536	15.607
crc32c byte-wise	64	11.396
	128	10.818
	256	10.405
	512	10.157
	1024	10.075
	1500	10.075
	2048	9.992
	4096	10.075
	8192	9.827
	9000	9.909
	16384	9.744

Table A.3: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
	32768	9.662
	65536	9.496
crc32c slicing-by-4	64	7.977
	128	7.06
	256	6.458
	512	6.103
	1024	5.929
	1500	5.88
	2048	5.83
	4096	5.789
	8192	5.764
	9000	5.764
	16384	5.747
	32768	5.756
	65536	5.813
crc32c slicing-by-8	64	5.904
	128	4.922
	256	4.137
	512	3.625
	1024	3.353
	1500	3.295
	2048	3.204
	4096	3.138
	8192	3.105
	9000	3.097
	16384	3.088
	32768	3.08
	65536	3.08
Adler-32	64	3.113
	128	2.585
	256	2.32
	512	2.172
	1024	2.106
	1500	2.106
	2048	2.073
	4096	2.056
	8192	1.99
	9000	1.99

Table A.3: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
	16384	1.974
	32768	1.957
	65536	1.965
Fletcher-32	64	2.882
	128	2.486
	256	2.263
	512	2.254
	1024	2.04
	1500	2.023
	2048	1.998
	4096	1.965
	8192	1.949
	9000	1.949
	16384	1.941
	32768	1.941
	65536	1.949
one's comp 16-bit word (in C)	64	1.759
	128	0.983
	256	0.678
	512	0.552
	1024	0.475
	1500	0.462
	2048	0.443
	4096	0.425
	8192	0.417
	9000	0.416
	16384	0.414
	32768	0.41
	65536	0.41
one's comp 32-bit word (in C)	64	1.594
	128	1.255
	256	0.974
	512	0.842
	1024	0.769
	1500	0.746
	2048	0.735
	4096	0.716
	8192	0.707

Table A.3: Continued...

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:
	9000	0.706
	16384	0.703
	32768	0.701
	65536	0.699
one's comp 64-bit word (in C)	64	2.246
	128	1.643
	256	1.272
	512	1.115
	1024	1.024
	1500	1.016
	2048	0.991
	4096	0.966
	8192	0.958
	9000	0.958
	16384	0.95
	32768	0.95
	65536	0.941

Table A.4: Comparative performance evaluation of different one's complement checksum implementations on Intel Core i3-2330M and Xeon with GCC optimization flag O1

One's complement checksum Algorithms	Data buffer size (in bytes)	No. of CPU cycles per byte:	
		Intel Core i3-2330M	Intel Xeon
one's comp 16-bit word (in C)	64	2.09	1.68
	512	0.837	0.733
	1024	0.755	0.661
	1500	0.72	0.632
	9000	0.668	0.579
one's comp 32-bit word (in C)	64	1.12	1.13
	512	0.504	0.437
	1024	0.412	0.367
	1500	0.392	0.335
	9000	0.343	0.297
one's comp 64-bit word (in x86_64 assembly)	64	1.08	0.967
	512	0.353	0.304
	1024	0.301	0.26
	1500	0.287	0.251
	9000	0.26	0.222

Table A.5: Effect of adding extra bytes to the buffer on Intel CPUs when computing CRC32C using hardware instructions CRC32 and PCLMULQDQ

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of extra bytes added to the buffer:	No. of CPU cycles per byte:	
			Intel Core i3-2330M	Intel Xeon
crc32c (crc32 + pclmulqdq based recombination)	64	0	1.25	1.23
		1	1.35	1.29
		2	1.27	1.25
		3	1.53	1.32
		4	1.33	1.06
		5	1.28	1.1
		6	1.38	1.17
		7	1.65	1.33
	1500	0	0.211	0.168
		1	0.213	0.169
		2	0.213	0.169
		3	0.214	0.171
		4	0.213	0.167
		5	0.217	0.201
		6	0.211	0.168
		7	0.214	0.17
	9000	0	0.151	0.128
		1	0.152	0.127
		2	0.151	0.128
		3	0.154	0.128
		4	0.154	0.128
		5	0.152	0.13
		6	0.152	0.129
		7	0.151	0.128

Table A.6: Effect of adding extra bytes to the buffer on Intel CPUs when computing TCP one's complement checksum with 16-bit word

CRC32C/ Checksum Algorithms	Data buffer size (in bytes)	No. of extra bytes added to the buffer:	No. of CPU cycles per byte:	
			Intel Core i3-2330M	Intel Xeon
one's comp 16-bit word (in C)	64	0	1.29	1.04
		1	1.34	1.09
		2	1.3	1.09
		3	1.27	1.08
		4	1.37	1.08
		5	1.37	1.08
		6	1.44	1.09
		7	1.35	1.09
	1500	0	0.275	0.227
		1	0.378	0.228
		2	0.274	0.226
		3	0.277	0.227
		4	0.268	0.222
		5	0.274	0.223
		6	0.27	0.237
		7	0.274	0.224
	9000	0	0.207	0.174
		1	0.21	0.174
		2	0.208	0.175
		3	0.209	0.18
		4	0.208	0.175
		5	0.208	0.175
		6	0.206	0.175
		7	0.205	0.177