2014

Performance Comparison of Projective Elliptic-curve Point Multiplication in 64-bit x86 Runtime Environment

Ninh Winson
Nova Southeastern University, winston@ninh.org

Performance Comparison of Projective Elliptic-curve Point Multiplication in 64-bit x86 Runtime Environment

by
Winston Ninh

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University
2014

We hereby certify that this dissertation, submitted by Winston Ninh, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

_____ _____

Wei Li, Ph.D. Date
Chairperson of Dissertation Committee

_____ _____

James Cannady, Ph.D. Date
Dissertation Committee Member

_____ _____

Junping Sun, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Eric S. Ackerman, Ph.D. Date
Dean, Graduate School of Computer and Information Sciences

Graduate School of Computer and Information Sciences

Nova Southeastern University

2014

An Abstract of a Dissertation Submitted to Nova Southeastern University in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Performance Comparison of Projective Elliptic-curve Point Multiplication in 64-bit x86 Runtime Environment

by
Winston Ninh

September 2014

For over two decades, mathematicians and cryptologists have evaluated and presented the theoretical performance of Elliptic-curve scalar point-multiplication in projective geometry. Because computation in projective domain is composed of a wide array of formulations and computing optimizations, there is not a comprehensive performance comparison of point-multiplication using projective transformation available to verify its realistic efficiency in 64-bit x86 computing platforms. Today, research on explicit mathematical formulations in projective domain continues to excel by seeking higher computational efficiency and ease of realization. An explicit performance evaluation will help implementers choose better implementation methods and improve Elliptic-curve scalar point-multiplication. This paper was founded on the practical solution that obtaining realistic performance figures should be based on more precise computational cost metrics and specific computing platforms. As part of that solution, an empirical performance benchmark comparison between two approaches implementing projective Elliptic-curve scalar point-multiplication will be presented to provide the selection of, and subsequently ways to improve scalar point-multiplication technology executing in a 64-bit x86 runtime environment.

Acknowledgements

I offer my sincere gratitude to my dissertation chair, Dr. Wei Li, who patiently guided me throughout this entire endeavor. I could not have completed this without his advice. I would like to thank my committee members, Dr. James Cannady and Dr. Junping Sun, who also provided me invaluable input throughout this process.

I would also like to thank my wonderful wife, Lan. She has always encouraged and supported me in my academic pursuits, often taking the reins on household matters (not to mention sacrificing many vacations and trips to the shopping mall) in order to allow me to focus on school work.

I would also like to acknowledge our two daughters, Kimberly and Amie, for spending countless hours pointing out awkward sentences and misspellings. I am so thankful for having such eloquent children.

Last but not least, I would like to express the utmost thanks to my adorable cat, Timmy, who spent many late nights cuddling around my feet and keeping me company while I worked on this dissertation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

For over two decades, mathematicians and cryptologists have evaluated and presented the performance of Elliptic-curve scalar point-multiplication in projective geometry using two basic quantitative metrics: the total number of multiplications (M) and squarings (S). Although these two single-digit mathematical operations are necessary to complete the multiplication of a scalar value k and a point p with coordinates (x, y) on an Elliptic curve, the question remains whether they are really sufficient to provide proper selection between projective Elliptic-curve scalar point-multiplication. Such questionable sufficiency in evaluating performance using single-digit M and S metrics, without accounting for optimizations and the cost of modulo arithmetic, will remain theoretical and unrealistic. Therefore, the performance result will not reflect the true figure between different projective transformation technologies.

This research will center on the performance comparison between two Projective Elliptic-curve Point Multiplication Agents (PEPMA) software: One was implemented in Network Security Services (NSS, 2013) and the second in (OpenSSL, 2013). Both NSS and OpenSSL have been deployed in the field to target a wide range of applications. Nevertheless, given the variety of projective transformations, diversity of underlying arithmetic optimizations (NIST, 2010), and different computing platform architecture, an unanswered question is whether there is a way to select a faster one, or to improve PEPMA's efficiency based on an empirical comparison.

This chapter provides an introduction to research involving the evaluation of performance and the performance comparison of projective Elliptic-curve Point Multiplication in a 64-

bit x86 run-time environment (Kasper, 2012; Levinthal, 2004). This research contains the most relevant information which supports the preparation of essential evaluation software tools to address the unanswered questions (BOCHS, 2013; PAPI, 2013). It further elaborates the significance of research and provides a discussion of the issues. The investigation advocates the need for research on an enhanced-accuracy performance comparison of Projective Elliptic-curve Point Multiplication Agent, or PEPMA.

The goal of this investigation is to develop a formal evaluation methodology which will provide a practical approach to selecting higher-performance based on precise and accurate quantitative computational metrics. This research will address implementation differences between NSS and OpenSSL, present connectivity between mathematical modules (Blake, 2001), and explore weaknesses with current performance evaluation methods. Subsequently, the selection approach based on a formal evaluation methodology will provide definitive, repeatable and quantitative means to improve new designs or existing implementations of PEPMA.

The principles discussed below will provide a means to achieve formal evaluation methodology.

Projective Elliptic-curve Point Multiplication Preliminary

PEPMA is an efficient mathematical procedure (NIST, 2010) to compute a product of a scalar k with an affine coordinate (x, y). In order to produce the result $k\times(x, y)$, PEPMA must take into its functional equations several additional parameters besides k, x, and y (Koc, 2009; Certicom Research, 2009; ANSI, 2005; ANSI, 2001). Additional parameters include, but not limited to, Elliptic-curve coefficients a, b, and the modulus m for modular arithmetic. PEPMA normally works under a Public-key Exchange protocol

(PKE). One available PKE protocol is Elliptic-Curve Diffie-Hellman (ECDH), where most parameters required for PEPMA are usually taken from a public certificate, subcategory "domain parameters". The ECDH Public-key Exchange protocol processes the scalar product $k(x, y)$ outputting from PEPMA to generate cryptographic private keys for data encryption or decryption (IASE, 2013; NIST, 2007). Typically, PEPMA will position itself in a cryptographic service hierarchy as shown in Figure 1.

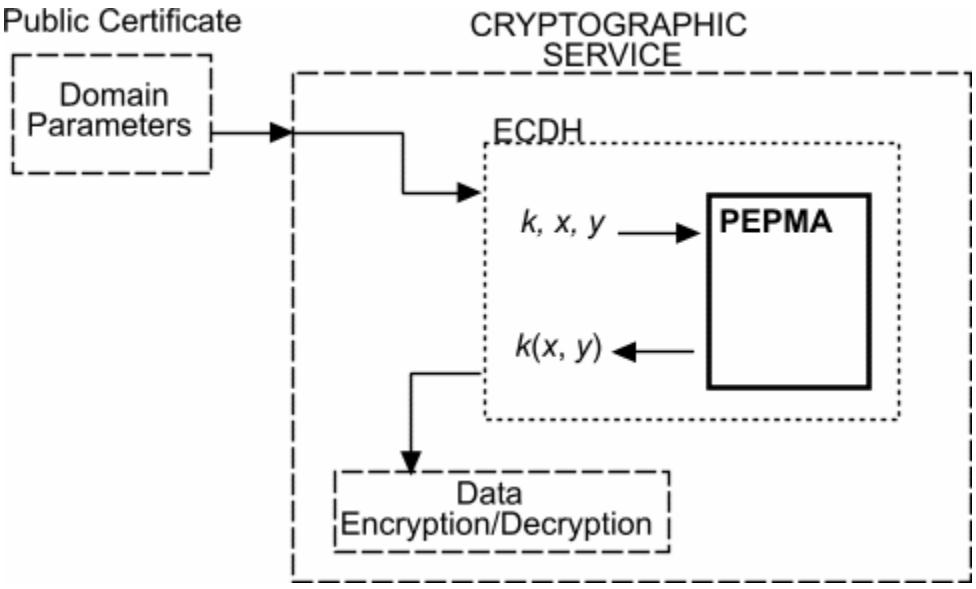

Figure 1. Position of PEPMA in a Cryptographic Service Hierarchy

Performance comparison and improving PEMA efficiency begins with a root understanding of point-multiplication in the projective domain (NIST, 2010; Hankerson et al., 2004; Menezes et al., 1996). In Figure 2, the scalar value k and the affine coordinates (x, y) of an Elliptic-curve point p enters ①. These entrant parameters to the projective transformation Elliptic-curve Point Multiplication (EPM) are 521 bits in length. At ②, the affine input parameters (k, x, y) are transformed into the projective

coordinate system simply by attaching Z=1 to the coordinates *x* and *y*. Chapter two will further explain why this attachment is valid in a finite field.

For representation purposes, the coordinates are designated as (X, Y, Z), and the first projective coordinate to enter the computational loop ③ has a value of (X = *x*, Y = *y*, Z = 1). A more detailed discussion can be found in Chapter two.

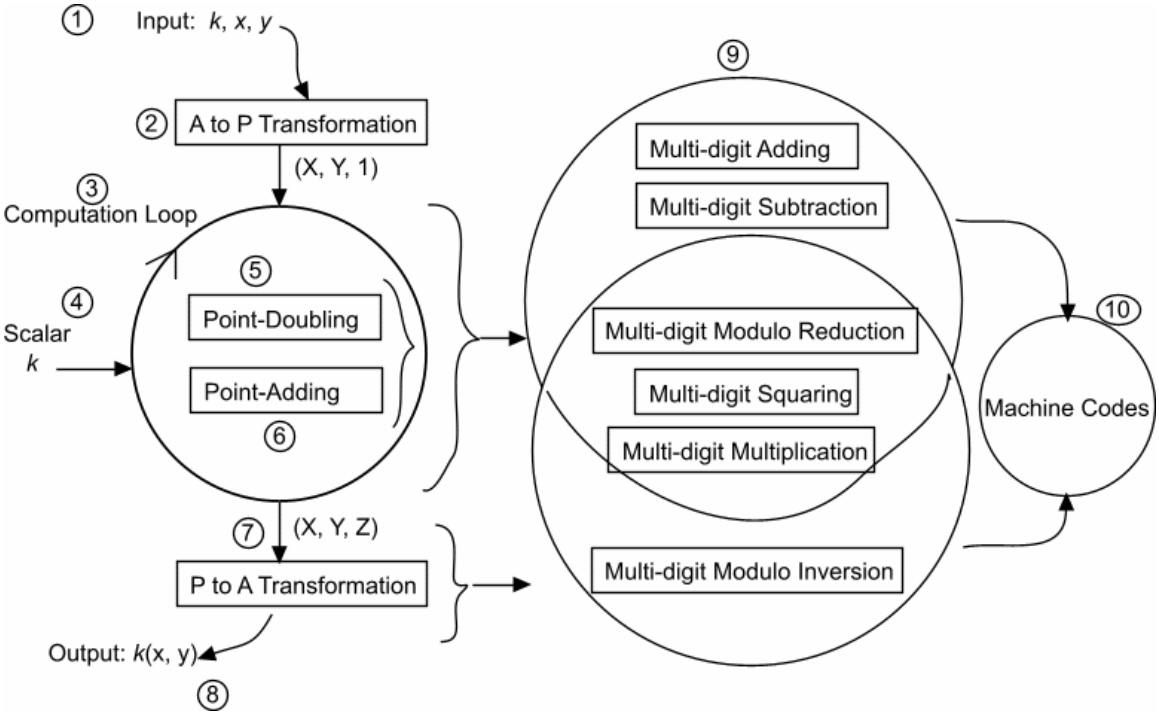

Figure 2. Projective Elliptic-Curve Point-multiplication Agent

The scalar *k* will control the number of point-doubling ⑤ and point-adding ⑥ operations in the computation loop ③. Operations in ③ are commonly designated as exponentiation procedures for PEPMA. The computation looping ③ will call functions ⑤ and ⑥ a few hundred times to produce the final result (X, Y, Z) at ⑦.

Efficiency in terms of how many times point-doubling or point adding is required to execute depends on the exponentiation algorithm used: left-to-right binary-shift, right-to-

left binary-shift, left-to-right fixed-base windowing-shift, or right-to-left fixed-base windowing-shift (Brown et al., 2001). Both NSS and OpenSSL use the right-to-left fixed-base windowing-shift exponentiation method. These methods have been frequently discussed (Saldamli et al., 2009; Avanzi, 2004; Koblitz, 2000; Cohen et al., 1998).

At ⑦, the "Projective to Affine Transformation" procedure converts the final projective coordinates (X, Y, Z) back to the affine coordinates at ⑧. The result $k(x, y)$ will be the multiplication of a scalar k with an Elliptic-curve point p having two affine coordinates (x, y).

All mathematical routines shown in ⑤, ⑥, and ⑦ call for multi-digit modulo arithmetic with the chosen field-modulus m (NIST, 2010). The Elliptic-curve Point Multiplication (EPM) mathematical services recommended in the NIST Suite B cryptography prime field suggests that a complete 521-bit big-number in a 64-bit system can be efficiently stored in nine 64-bit registers using $9 \times 64 = 576$ bits (NSA, 2013). However, both NSS and OpenSSL represent the big-numbers differently from the nine 64-bit registers with arithmetic carry bit. The notation of big-numbers in Chapter two will further describe the format, differences, advantages, and disadvantages between the NSS and OpenSSL representation of multi-digit numbers.

At the multi-digit arithmetic ⑨, six big-number arithmetic operations are required to support PEPMA: adding, subtracting, modular reduction, squaring, multiplication, and inversion (Certicom Research, 2009). Except for modular inversion, all five operations are necessary for point-doubling and adding in the computing loop ③. To convert projective coordinates back to affine coordinates, one or two modular inversions along with adding, modular reduction, and squarings are required in block ⑦. Since block ⑦

"P to A Transformation" is located outside of the loop and executed only once at the end, its computing cost is low compared to the cost of point-doubling and adding while in the computational loop ③.

All arithmetic in block ⑨ will be compiled into machine codes, as shown in block ⑩. Computational costs of projective EPM at block ⑩ can be documented by examining the assembly codes produced by the target C compiler. The NSS code in Appendix A further details this process.

Point Doubling and Point Adding

The Elliptic-curve Point Multiplication procedure (EPM) requires two functions working together in the exponentiation loop: point-doubling of a point and point-adding of two different points (Certicom Research, 2009; Cohen et al., 2006; Connel, 1999). For example, let $p(x, y)$ be an affine point on an Elliptic curve. Let k be a scalar multiplied with point p. If $k = 5$, then to obtain $5 \times p$ efficiently, two point-doublings and one point-adding are applied:

$$k(x, y) \triangleq 5 \times p \triangleq [2 \times (2 \times p)] + p$$

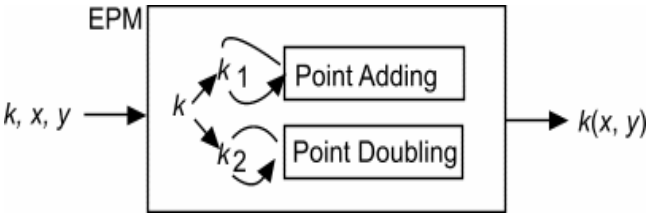

Figure 3. Elliptic-Curve Point-multiplication in Affine Coordinate

The efficient affine-coordinate mathematical operations above require exactly two point-doublings and one point-adding, while k controls which function to use and how many

times to call them. In other words, the exponentiation of p has occurred 2 times in the exponentiation loop [$2 \times (2 \times p)$] while the adding of p has occurred once.

There are several ways to construct software servicing the scalar product $k(x, y)$ in 64-bit computing platforms (Avanzi et al., 2006; Fong et al., 2004; Koblitz, 2000). One method applies the time-domain computation to affine coordinates in a finite field (Koc, 2009). Based on algebraic laws, $2 \times (x, y)$ is equivalent to the point-doubling of point $p(x, y)$ on an Elliptic curve (EC). Point-doubling arithmetic will produce a result in another point $p_3(x_3, y_3)$. The coordinates of this resultant vector are precisely defined by two Cartesian coordinate equations in the Euclidian plane. A derivation of these formulas can be found below, and in (Blake, 2001):

$$x_3 = \left(\frac{3x^2 + a}{2y}\right)^2 - 2x, \qquad y_3 = \left(\frac{3x^2 + a}{2y}\right) \times (x - x_3) - y$$

The parameter "a" is defined as a domain coefficient of an Elliptic curve. The selection of "a" has been chosen carefully by cryptologists for computational ease, and at the same time, to satisfy important security criteria. Coefficient "a" is set to –3 per NIST recommendation for implementation of a P-521 curve. NIST defined and explained these settings in (FIPS PUB 186-4, 2013) and (NIST, 2010).

Precise modulo arithmetic must be applied after each arithmetic operation whenever there is an arithmetic overflow beyond the chosen boundary of finite field F. In calculating coordinates x_3 and y_3, take the inverse of $2y$

$$r \equiv \frac{1}{2y} \mod m$$

by following the inversion rule (Ciet et al., 2006):

$$y \times r \equiv 1 \mod m \qquad (1.1)$$

To derive the inversion of y, PEPMA might need to search for one unique value of r in the entire finite field having $2^{521} - 1$ elements for which equation (1.1) is satisfied. This operation will be computationally intensive (Ciet et al., 2006; Itoh et al., 1988). However, the calculation of the scalar product $2 \times p(x, y)$ will be faster if the inversions of y can be eliminated, or at least significantly reduced from a few hundred to one or two times in the $k\times(x, y)$ loop.

Despite the power of modern-day computing platforms, the current embedded processors and RISC in tablets have limited arithmetic capabilities to process Elliptic Curve Point Multiplication in a timely manner (ARM, 2013; ZigBee, 2010; Jennic JN5184, 2010). Computation using affine coordinates will require significant longer time, due to the lengthiness to compute inversions. Therefore, realization of time-domain EPM in these limited arithmetic capability processors will not be practical. Under the finite projective theory, the elimination of inversions can be realized by transforming the affine coordinates (x, y) into projective coordinates and processing the computation of scalar product $k(x, y)$ entirely in the projective domain. Computation in projective coordinates found in Chapter two will further explain this realization.

When a projective transformation is activated, a forward Affine-to-Projective Transformation (APT) converts affine input parameters k, x, y to parameters with their representations in projective domain. After point-adding or point-doubling functions complete their mathematical operations entirely in projective domain, the reverse Projective-to-Affine Transformation routine (PAT) converts the result back into its equivalent affine coordinates, $k(x, y)$. This concept is recorded in Figure 4, and in (NSS PEPMA, 2013; OpenSSL PEPMA, 2013; Cohen et al., 2006).

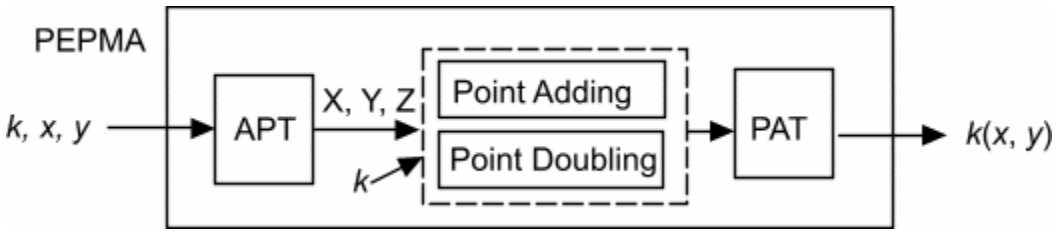

Figure 4. Elliptic-Curve Point-multiplication in Projective Coordinate

When implementing PEPMA to work efficiently under weighted projective transformation, also known as transformation of variables into Jacobian's domain (Koc, 2009), there will be exactly one 521-bit inversion in the PAT and none in the APT. The repetitive mathematical operations in point-adding or point-doubling functions do not have any inversions. The point-adding and point-doubling functions described in Figure 4 above will be processed entirely in the projective domain. Their mathematical operations will no longer be associated with affine coordinates after an Affine-to-Projective Transformation (Bernstein et al., 2007; Ryabko et al. 2005).

There are different ways to compute projective point-doubling or point-adding functions; but yet, the product $k(x, y)$ will be the same at the end (Cohen et al., 2006; Brown et al., 2001). This raises an issue of interoperability between these computing approaches. Can point-doubling or point-adding functions be mixed and matched? Which one is better in terms of efficiency? An immediate question is whether the performance of PEPMA is unknown based on existing theoretical work.

Associated Environments of PEPMA

The environments surrounding PEPMA will potentially affect the runtime performance of PEPMA. These environments include system architecture, compiling options, and runtime domain parameters.

System Architecture

The chosen system architecture for PEPMA will limit how a big-number or Multi-Digit Number (*MDN*) can be represented efficiently. Testimony from researchers indicated several ways to represent an *MDN* contained in a finite field *F* (GNU-MP, 2011; SEC 1, 2000). However, only two types of representations are commonly used in the industry: Prime field F_p and exponential prime field F_p^s.

If prime *p* is set to 2, then the exponential prime field F_p^s becomes F_2^s, or an exponential binary field. Moreover, if the *MDN* is implemented using a two-bit field F_2, then NSS or OpenSSL PEPMA can represent an *S*-bit Multi-Digit Number as a finite discrete polynomial along with a *sign* indicator

$$MDN = (sign)\left[b_{S-1}(2^{S-1}) + b_{S-2}(2^{S-2}) + ... + b_1(2^1) + b_0(2^0) \right]$$

Each arithmetic digit in 64-bit system architecture can hold 64 bits plus a carry bit. Effectively, a full digit contains 65 bits. Since NSS uses a half-digit representation (32 bits) and OpenSSL uses a 58-bit representation (also called field element or *felem*) instead of 64 bits architecture, a question that comes to mind is which method would be more efficient.

Compiling Options

Users compiling options have several levels of optimization to choose from (GCC, 2013). For example, optimization switch –O0 in a GCC compiler will turn off all optimizations while an –O1 option will turn on some optimizations. Another facet to explore is whether optimizations affect the cost index and what setting would work best for computational efficiency.

Run-time Domain Parameters

The binary content of vectors coming from domain parameters is expected to contribute to the performance evaluation of PEPMA. Both NSS and OpenSSL PEPMA work under an Elliptic-Curve Diffie-Hellman (ECDH) Public-key Exchange protocol to generate cryptographic keys for data encryption and decryption.

The ECDH cryptography protocol used for exchanging private keys is believed by researchers and industry professionals to provide a secured transaction under an unsecured communication channel. One area of concern in evaluating PEPMA's performance is why, where and how domain parameters affect the assessment. A further examination of ECDH protocol might help in this regard. For a more detailed transaction of ECDH protocol and associated domain parameters, readers are referred to the contents of Appendix B and Appendix C.

Under public viewers and on an unsecured communication channel, the calculations calling for PEPMA's services in transaction sequences ① to ⑤ are summarized in Figure 5 below. More details descriptions of the Elliptic-Curve Diffie-Hellman key exchange protocol (ECDH) are found in (NIST, 2007), NIST Special Publication 800-56A.

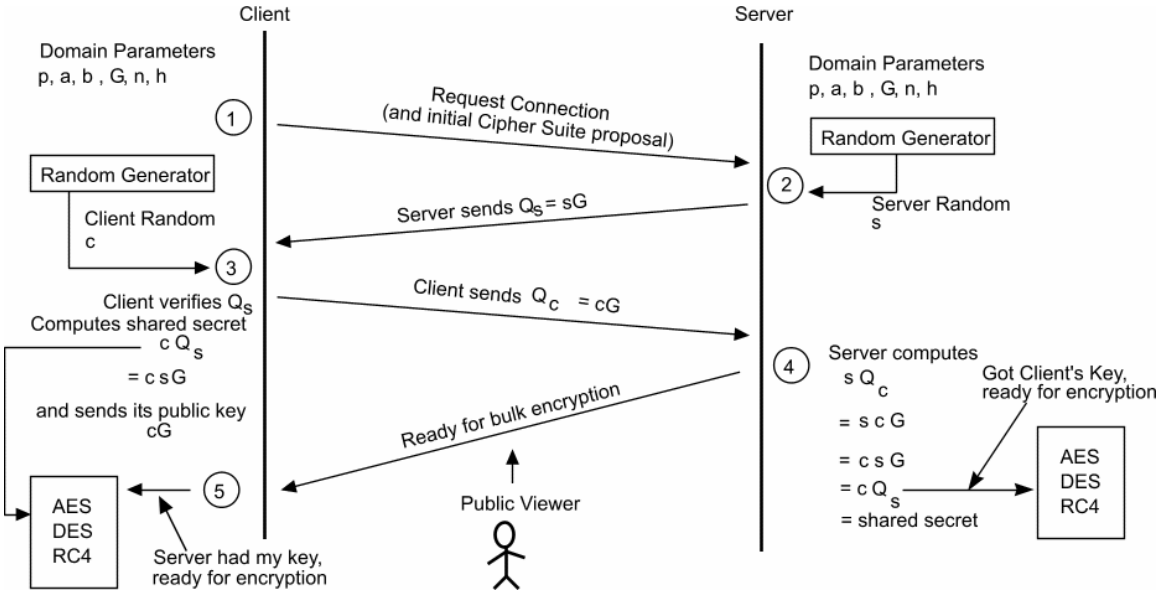

Figure 5. Elliptic-Curve Diffie-Hellman Key Exchange Used with PEPMA

The Client's ECDH procedure initiates transaction ① starting with Client's domain parameters (p, a, b, G, n, h)[1]. Subsequently, the scalar product calculations of $k(x, y)$ provided by PEPMA occur at the computations of sG, cG, csG, and scG, where G(x, y) is the generator[2] for the cyclic subgroup within the chosen finite field. At transaction ④, the Server receives Client's key and is ready for data encryption using Advanced Encryption Standard, AES (FIPS-197, 2001), or Data Encryption Standard, DES, or Rivest Cipher 4 (RC4) encryption algorithm for streaming data.

---

[1] The ECDH transaction with numerical details and definitions of domain parameters are recorded in Appendix B.

[2] The generator for the cyclic subgroup is a point on Elliptic-curve where the result of the product nG(x, y) equals to a point at infinity. This generator is also known as the base point of the cyclic subgroup.

The Empirical Performance Evaluation

To acquire parametric pertaining to performance and to compare the computational efficiency of PEPMA in a 64-bit x86 run-time environment, two optimizing codes will be selected for investigation: one made in Network Security Services (NSS, 2013) and the other from the OpenSSL Project (OpenSSL, 2013). Both projects have core implementations of PEPMA recommended in the NIST Suite B cryptography under prime field (NSA, 2013). The NSS and OpenSSL PEPMA are among the first industry open-source applications to implement and deploy an NIST public key exchange with 521-bit Elliptic-curve cryptography. In order to achieve higher efficiency, both NSS and OpenSSL 521-bit prime-field implementations applied the weighted projective transformation, or the transformation of variables into Jacobian's domain.

The NSS and OpenSSL provide free source codes of cryptographic low-level implementation, along with high-level implementation protocols. The NSS libraries currently service cryptographic functions for Firefox, Android, and other applications that require Public Key Exchange services. The OpenSSL currently serves a majority of consumer products, such as embedded TCP/IP cameras, home desktop videos, and smart TVs.

Both NSS and OpenSSL have received a variety of FIPS-140-2, security level 1, 2 or 3 certifications indicating that the implementations are adequately stable (FIPS-140-2, 2001). The codes can be applied to Elliptic-curve public key exchange cryptography to ensure authenticity in the public-key infrastructure.

Open-source projects can offer an exceptionally important role in benchmarking. A particular FIPS certified implementation that has gone through thorough testing by a certification and accreditation agent might provide a well-defined baseline for

comparison. Cryptographic Algorithm Verification Program (CAVP, 2013) also describes the verification procedures and provides additional information. Without this reference for comparison, it might be difficult if not impossible to present any valid performance evaluation by counting the number of mathematical operations as often claimed in current literature. This is a primary motivation for deriving a comprehensive performance comparison between NSS PEPMA and OpenSSL PEPMA, all operating in a 64-bit x86 run-time environment.

Additionally, the 64-bit x86 computing architecture available today is becoming popular computer platform; hence, obtaining comparative performance figures based on these specific computing platforms with accurate cost metrics will have immediate commercial benefits. Such explicit performance evaluation will help crypto software developers to choose an effective projective transformation method which contain efficient underlying mathematics for the realization of a Projective Elliptic-curve Scalar Point Multiplication Agent.

It has been suggested by (Pare, 2004; Gillham, 2003; Yin, 2003; Yin 1994) that the empirical performance evaluation based on case studies will be well suited to answer the questions on PEPMA's topic such as: "Is performance of PEPMA unknown even based on existing theoretical work; Or, what are the metrics to truthfully evaluate PEPMA's efficiency?"

Latency of PEPMA

Many researchers have used the computational unit for multiplication based on a full-word mathematical procedure. The computational unit does not account for the cost of a digit-by-digit (or limb-by-limb) operation and is counted as 1M in literature (M =

Multiplcation). For example, if an operand Elliptic-curve key length is 521 bits, then a full-word hardware multiplier operates a multiplication of 521-bit word by 521-bit word operands simultaneously, and immediately produces a 1042-bit result in a single multiplication. This 521-bit "*single- shot*" multiplication is currently not available in any general CPU. This lack of "*single shot*" multiplication compounds the latency evaluation. Thus, in order to practically determine the latency of PEPMA, the measurement unit "M" should at least be converted to computational cost based on digit-by-digit multiplication. Furthermore, the latency evaluation becomes even more complicated in NSS and OpenSSL 64-bit processing where each digit in a target CPU could be any arithmetic word length: 8, 15, 16, 22, 32, 56, or 64 bits with or without hardware carry bit.

Also, the latency of PEPMA affected not by one, but by at least two hardware components: Arithmetic unit integer quad-word multiplication with *imulq* instruction and memory utilization with quad-word memory move, *movq* instruction. Both NSS-PEPMA and OpenSSL-PEPMA executable codes use a significant number of *movq* instructions (Intel Latency, 2013). While the latency index of *movq* and *imulq* instruction is 6 and 10 respectively, the multiplication routine *s_mpv_mul_d_add*() in NSS PEPMA executes a total of 29 *movq* instructions and only 4 *imulq* instructions (Intel Latency, 2013). Table 66 in Appendix A lists out the routine *s_mpv_mul_d_add*().

With the same memory utilization subject, literature from (Singhal et al., 2011; Levinthal, 2009) provides some guidance for reading and applying memory utilization factors as an intricate part of the performance analysis. Given these two hardware dependencies, it is difficult to extrapolate from academic findings. It is exceedingly

difficult to construct a vector test set without reference implementations because latency will vary substantially by the test vector's content.

In order to address the complexities of performance improvement of PEPMA, one needs to determine how academia and industry have tried to evaluate PEPMA in terms of computing costs.

Coarse Estimates Efficiency of PEPMA

Counting mathematical operations with Multiplications (M) or Squaring (S) at the top level of PEPMA service routines offer coarse estimates. To address the performance issues quickly and more precisely than coarse estimates, the researchers often rank software latency with a single metric using *clock*() time function (See Appendix I), which is readily available in common computing platforms (GNU-CPU-Time, 2014).

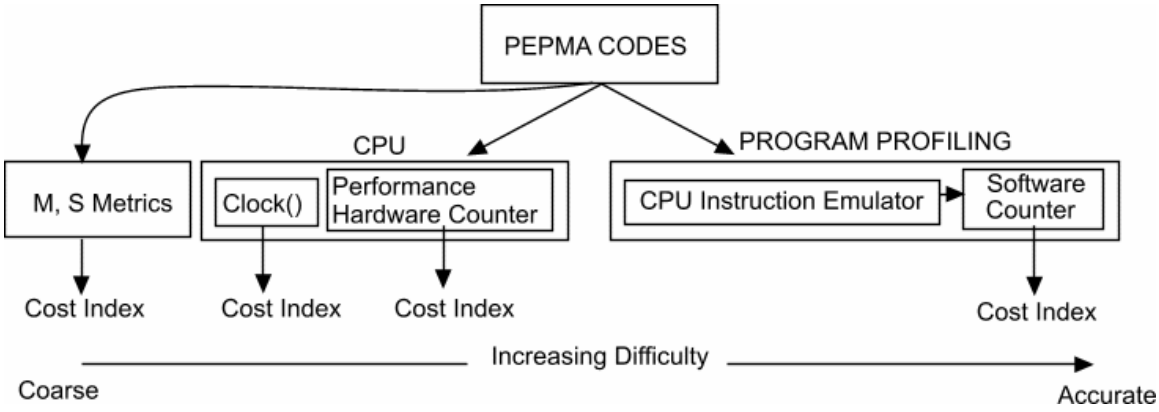

Figure 6. Performance Measurement Techniques

Although the estimated cost using a computing platform's *clock*() function offers a quick evaluation of performance, it lacks insights into the internal structure of PEPMA; thus, these cost indexes do not furnish any useful information for improvement along the computational chain. The M, S and *clock*() metrics for obtaining rough cost indexes are

shown in Figure 6. Operational difficulty spans from easy on the left to more difficult on the right.

Ranking the efficiency with a CPU cycle counter *clock*() under a run-time environment, as shown in Figure 6, will help approximate the overall performance of PEPMA. However, the result will not be accurate and precise due to Operating System (OS) overhead, active running treads, and other processes running in the same runtime environment.

Academic work comparing PEPMA by the ratio of one processing runtime to the other runtime in terms of CPU clock cycles appeared in eBACS (eBACS, 2004) and from researchers (Bernstein, 2007; Somani, 2010). The eBACS performance evaluation was an eight year European research initiative launched in February 2004. In 2007, the group posted a web page where it tabulated a processing runtime of an Elliptic-Curve Diffie-Hellman key exchange procedure (ECDH) 256 key-pair generation without precomputation over GF(p) (Crypto++, 2007). This evaluation model has since been popular. If one decided to use PEPMA right out of an open-source repository, he or she knows right away whether the processing time can fit well into computing architecture.

While such single-unit performance-measurement process is mostly intuitive, there is an abundance of hidden features in the computing chain that can drastically change this performance measurement. Several hidden features can be spotted by systematically examining the NSS and OpenSSL PEPMA codes. Two particular features are noted at the exponentiation procedure where a number of projective doubling/adding functions can be reduced by the order of computations. Another hidden property is located in an NSS half-digit 32-bit numeric representation (NSS-1, 2013). However, the conversion of existing

codes from a half-digit 32-bit representation to a 58-bit or full-digit 65-bit numeric representation is possible in a 64-bit x86 system (IA-64-32, 2013, Section 4.2). Such successful conversion can significantly change the outcome of a single-unit performance-measurement. Hence, a PEPMA procedure can be improved using better evaluation metrics.

Fine Estimates Efficiency of PEPMA

Better performance evaluation of PEPMA available in cryptographic communities can be classified into two categories: performance measurement (PAPI, 2013; Levinthal, 2009; Drongowski, 2008) and program profiling through emulation (BOCHS, 2013; Code XL, 2013). The instrumentation setup in an efficiency measurement process might include one or two on-chip machine-code instruction hardware counters counting the occurrences of instructions. For example, operations MUL, the number of multiply operations executed by PEPMA, has resulted in an event 0x12, mask 0x00 in performance monitoring processing unit (IA-64, 2013). The Performance Hardware Counters sit inside CPU hardware. Their position related to PEPMA code is shown in Figure 6.

Almost all 64-bit x86 systems, including Intel Pentium and AMD processors made for PC/Servers, have incorporated two on-chip 40-bit performance hardware counters, which can be used to collect execution times of cryptographic service routines (Intel PERC, 2013). The performance program profiling through the emulation of a cryptographic program like PEPMA is available from Wind River SIMICS (WindRiver SIMICS, 2013) and from an open-source repository (BOCHS, 2013).

In 2008, a more elaborate performance comparison between cryptographic algorithms was performed on Intel XScale architecture (Bartolini et al., 2008). Bartolini et al. used an XScale computing platform (Intel XScale, 2007) as a reference processor and Multi-precision Integer and Rational Arithmetic (MIRACL, 2013) C library to construct 571-bit large integers during the evaluation. This signified that PEPMA required more accurate performance comparison and that coarse estimates will not suffice.

Targeting open-source software like OpenSSL, Google Corporation has been working on displaying performance tables and charts using a set of metrics such as benchmark machines, cycles per operation, and iteration counts for algorithms (Kasper, 2010). Kasper targeted the performance evaluation applied toward the Transport Layer Security (TLS) protocol (NSS-1, 2013) with shorter key-length (224 bits) NIST P-224 Elliptic-curve under prime field. The NIST P-224 mathematical procedure produces 224-bit crypto keys versus 521 bits in this evaluation (NIST, 2010). This indicated that another way to improve measurements is to use program profiling technology.

Program Profiling through Virtualization and Emulation

A virtual machine is a software engine that redirects code and data of an application and executes it within a newly created and isolated runtime environment. The VMware or VirtualBox by Oracle performs this function well (VirtualBox, 2014). Thus, virtualization refers to technology that provides an additional layer of glue-logic and services between hardware and PEPMA as an application. Different type of technologies can be used to employ virtual machines. The two most commonly used are direct execution with CPU instructions for fast speed, and emulation of CPU instructions for flexibility. Virtualization by emulation of PEPMA coding increases flexibility in terms of

obtaining computing costs (Mihocka & Shwartsman, 2014) Within the virtual environment, the emulation of PEPMA codes will allow precise and accurate counting of frequently used instructions such as imulq or movq (Intel Latency, 2013). Readers are referred to Appendix A for an accurate counting a small sample of NSS code (actual count will be in the order of million units). This performance measurement technology is referred to as *program profiling* of a PEPMA procedure and the units of measurement can be any CPU instructions (machine code). As a result of units of measurement like MULq or MOVq, counting these executing instructions with instruction emulation will be exact. Subsequently, computational cost equations from these units of measurements can be made. Under a particular emulation environment with a Community Enterprise Operating System, CentOS, a Linux OS, PEPMA will position itself in a software service hierarchy as shown in Figure 7.

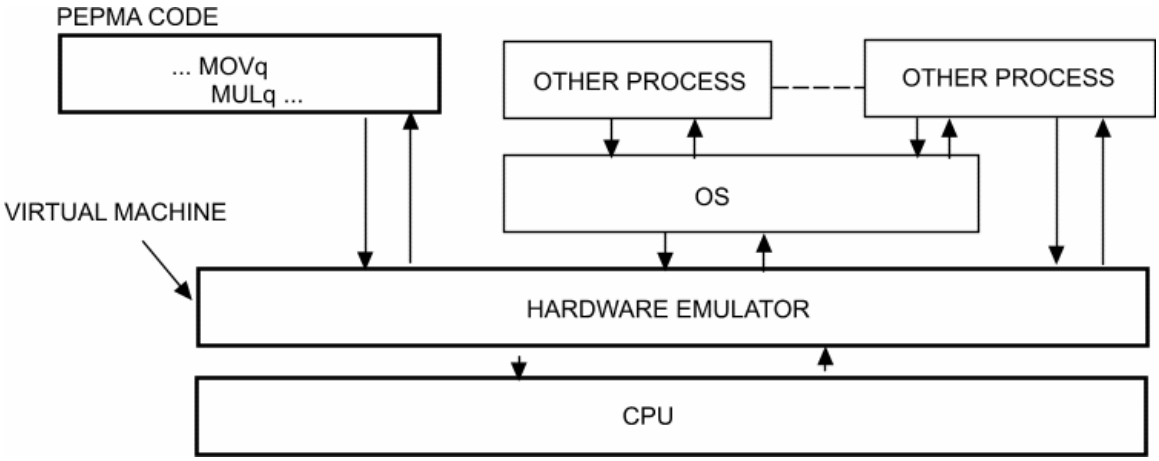

Figure 7. Program Profiling and Emulation of PEPMA

Accurate and Precise Efficiency Evaluation of PEPMA

When observing PEPMA as a mathematical solver, one will uncover a number of performance deficiencies during implementation due to misused algorithms, inefficient numerical representation, or platform dependency. According to Mittelmann (2004) and

other professionals (COCO, 2014; BBOD, 2013) in the field of benchmarking of linear optimization software[3], comprehensive benchmarking of each part of the solver will help identify potential efficiency problems, and will lead to software improvements. To benchmark each part of the solver, metrics M, S, *clock*(), Performance Hardware Counter and Software Counter can effectively provide input to the formulation analysis for verification of the result. More precise and accurate cost index can be derived from formulas instead of from other coarse cost indexes. The concept for verification is shown in a diagram in Figure 8.

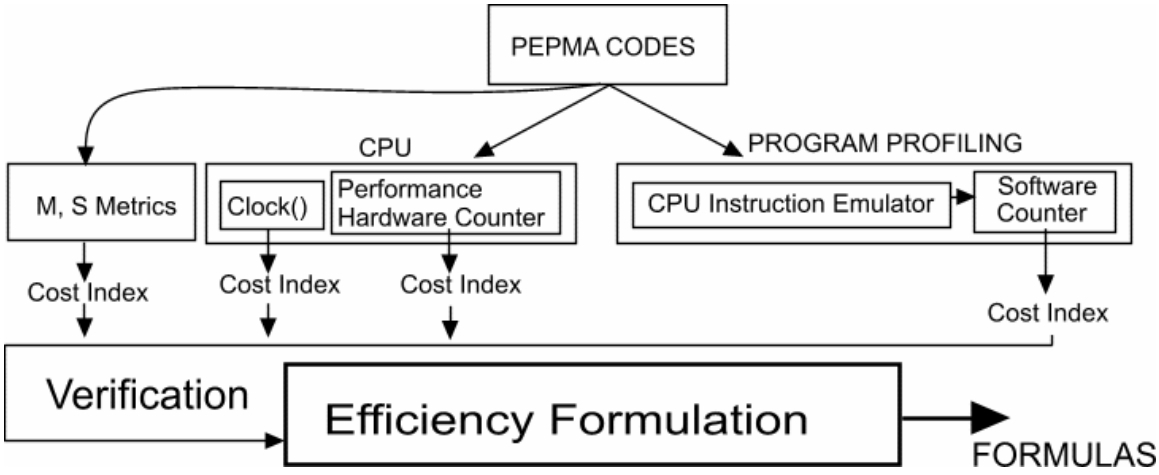

Figure 8. Accurate and Precise Efficiency Evaluation of PEPMA

Motivation and Direct Application

Of particular interest is the result obtained from applying PEPMA toward private key generation, specifically the outcome of the Elliptic Curve Diffie-Hellman Public Key

[3] Linear optimization is a method to achieve the best outcome.

Exchange protocol (IASE, 2013; NIST, 2007). Today in the Public-Key Infrastructure, these protocols are commonly used in many client-server transactions, and PEPMA may eventually be the dominant method for public key exchange in cyberspace security in the near future. This comparison topic was selected because PEPMA's fast response is an important factor in client-server transactions. Furthermore, obtaining maximum efficiency offers superior advantages in terms of shorter user waiting times, even in less powerful processors.

Problem Statement, Goal and Objectives

Problem Statement:

Presenting the computational performance of Elliptic-curve scalar point-multiplication approaches in projective geometry using:

(a) the total number of single-digit non-modular multiplications (M) metric,
(b) the total number of single-digit non-modular squarings (S) metric,
(c) or executing computations under unspecified underlying arithmetic methods,
(d) or under an unspecified computing architecture

to complete the multiplication of a scalar value k and a point p with coordinates (x, y) is necessary but insufficient.

Goal:

Given a mixture of projective transformations, diversity of underlying arithmetic algorithms, and different computing platform architectures, the goal of this research is to improve the projective Elliptic-curve point-multiplication agent by dynamically or statically selecting higher-performance arithmetic approaches based on quantitative computational metrics.

Objectives:

In order to provide a higher-performance approach between PEPMA, additional quantitative performance figures of merit will be introduced.

Barriers and Issues

Qualifications of Quantifiable Metrics

Qualifying metrics will be the most difficult part of this research. Challenges come from determining quantifiable cost indexes for each mathematical procedure in the computing chain of PEPMA. An initial investigation showed that the contributing complications to the cost index might include optimizations in selected algorithms, presentation of big numbers, values of chosen field modulus, test vectors, base points, and mathematical optimization factors.

Mathematical Optimization Factors

In order to construct accurate metrics, optimization factors must be included in the equation. The contributing complexities to mathematical optimization include operators such as bypassing functions based on special vector contents, early exiting executing loops, modulo reduction methods, and the dynamic selection of computations via other transformations or dynamic shortcuts such as a function of input vectors.

Selection Criteria

The criteria of selection for better efficiency will pose other challenges as well. These challenges come from constructing of practical Key Performance Indicators based on quantifiable metrics of various PEPMAs.

Mathematical Traceability

There is concern regarding the qualification of cryptographic mathematics and underlying low-level implementation of mathematical routines. Many original concepts, underlying theorems, formulations and properties described will be introduced in the context of Elliptic-curve point-multiplication, finite-field projective geometry, and 521-bit prime-field arithmetic. Rigorous proofs for these theorems may be found in Elliptic-curve and projective transformation literature (NIST, 2010; Certicom Research, 2009; Blake, 2001; Menezes et. al., 1996; Cohn, 1962) and Finite-Field Projective Geometry (Rosen, 2006).

Research Questions

This research will answer the following questions:

1) Is the performance of PEPMA unknown based on existing theoretical work?

2) What are the metrics to truthfully evaluate PEPMA's efficiency?

3) Are there ways to improve PEPMA’s efficiency based on the empirical comparison?

Relevance, Significance and the Need to Evaluate

Relevance:

In the fast-paced cyberspace security, threats to public key exchange are constantly emerging. One possible goal of improving security is to maximize the burden for adversaries in terms of computational costs, such that adversaries will not be able to retrieve or reveal private keys. According to US-CERT (2014), more than hundred thousands damaging intrusion attacks to the U.S. military network occurred in FY 2011 (OMB, 2012). This highlights the need for next-generation public-key exchange design to

have the capability to withstand brute-force cryptanalysis executing under high-performance, but low-cost computing platforms (Intel AVX, 2012).

Naturally, to meet these goals, usage of longer key length is necessary (NIST 800-56A, 2013; Barker et al., 2012). This requirement poses a major challenge to software professionals who will need to search for an innovative approach to derive private keys as efficiently as possible. Despite a large working community in cryptology mathematics, the foundation has only provided limited evaluation techniques for reducing Elliptic Curve (EC) computation (Hankerson et al., 2004).

In NIST Special Publication 800-57 (Barker et al., 2012), a recommendation of key size and algorithm was provided for Public-Key Infrastructure (PKI) users and infrastructure components (e.g. X.509, X.509 DoD certificates). Table 2-1 in 800-57 lists a time period NIST recommended for use of the ECDH PKE class. According to this table, the ECDH curve P-384 digital signature certificate has expired after 12/31/2010. If the use of the public key is expected to continue after certificate expiration, then all certificates should also expire at an earlier date than specified in the table. Since the ECDH curve P-384 has already expired, this recommendation signifies an urgency to implement PEPMA P-521 for higher security PKE. This urgency pushes the optimal design and implementation of PEPMA an immediately valuable tool.

Today, research on explicit mathematical formulations in projective domain continues to excel by seeking higher computational efficiency and ease of realization (EFD, 2013). However, because computation in projective geometry is composed of a wide array of algorithms and optimizations, there are different ways to construct the arithmetic under projective transformations to produce different results in terms of

computational efficiency as seen in the literature of (Cohen et al., 2006). Given choices between selecting a variety of techniques to implement the projective transformations and the diversity of underlying mathematics, a fundamental issue of software engineering remains to be the optimum solution to a projective EPM problem. The consideration by software developers is usually efficient algorithms combine with the ease of implementation, which can be easily verified through comprehensive quantitative or qualitative performance figures of merit.

Significance:

The evolution of the Internet toward a vast, ubiquitously connected society is imminent. Services of large devices, formerly placed on desks, have now become consumer small parts, providing continuous information, business transactions and personal entertainment. Thus, efficient PEPMA is a critical technology to ensure that personal computing devices can deploy security functions as fast as possible. At the same time, that technology must be operable with existing security certificates. The comprehensive benchmarking of each computational part, as shown previously, can help establish precise baseline performance. It’s important to understand the weaknesses and strengths of each service routine; be able to identify computational efficiency and provide ways for improvement; and track PEPMA's cryptographic performance over time as future computing platforms evolve with cryptographic instruction extensions.

The performance comparison of PEPMAs will offer an important role in the EPM literature pool by uniquely comparing two reference implementation codes. This detailed evaluation can set a direction for future research in which this field can be built upon for more efficient PEPMA, and can also be tailored to the underlying computing platform.

Be able to implement this concept to the entire chain of computing services, and not just selecting the best PEPMA available and then use it as is, is certainly worth the research effort. The comprehensive interpretation and analysis of the performance parameters generated by the benchmarking of each computing process will comprise the main technical issues addressed in this paper.

The immediate contribution that this study makes to the cryptology field is that the performance comparison will provide the implementers/technologist of PEPMA with tools and analysis to select the best methodology in order to minimize development time and maximize the efficiency requirements in a 64-bit x86 system. If interventions such as those previously explained were not presented, PEPMA efficiency in implementation may not be easily realized. Additionally, future research and development directions pertaining to the NIST 521-bit PEPMA might advance faster as a result of step-by-step formulations in this research study. Using the same approach as presented, the less powerful, but ubiquitous 64-bit embedded processor Advanced RISC Machines (ARM), used mostly in today's tablets and cell phones, might even benefit from the analysis and formulation with just a few modifications to the approach schematic and metrics. An ARM architecture description is currently accessible from (ARM, 2013).

After carefully digesting the comparisons between PEPMAs, one should be able to suggest the first systematic examination of the design, deployment, and operational challenges encountered by projective transformation over the years. This performance comparison will reveal a fundamental gap between theory and operational arithmetic costs particularly with the computing resource-constrained processors. It is believed that the insights gained from the evaluation can offer valuable input for the improvement of

the arithmetic chain either dynamically or statically, in the application toward scalar multiplication *kp* in 32-bit or 64-bit run-time environments.

The need to evaluate PEPMA with more precision and accuracy:

During reviewing the literature on PEPMA with the intention to investigate where the validity of this evaluation stands with respect to the current research, one noticeable point is that benchmarking a complex solver akin to PEPMA technology in the commercial sector is much different than in academic research, where the primary goal is to quickly verify a simple computing approach. Although most publications support only conceptual findings, the importance of academic research is evident. Furthermore, regardless of what services are required underneath, the efficiency metrics developed while examining the exponentiation function will provide a speedy gauge between projective exponent algorithms (see blocks 1-8 in Introduction, Figure 2). However, this evaluation model resembles the comparison between black-box software, which is not very meaningful in terms of improving the black-box itself. This situation has likely arisen due to the fact that PEPMA is a highly intellectual product solely based on its own multipart arithmetic merits; and thus, it is difficult to evaluate without a complete solution and additional metrics, or without specifying an exact computing architecture. As a result of dealing with such complexity, academic researchers tend to publish papers about efficient ideas, instead of publishing about what is actually required in a full, practical implementation setting.

Lacking the support of concrete literature in processing PEPMA poses major obstacles in understanding intricate connections between computing modules; and thus, such dilemma

prevents improving software implementation or slowing down adapting services to the target computing hardware.

As of today, an evaluation of 521-bit key-pair generation with pre-computation over GF(p) with NIST Mersenne prime modulus (Solinas, 1999) is seldom found in academic literature or any industrial publication. This predicament exists because practical usage of such technology is just about to begin in both government and commercial sectors after a lengthy FIPS-140-2 certification and accreditation of the implementation. So far, there has not been public availability of this comprehensive performance comparison pertaining to PEPMA in a 64-bit x86 runtime environment. The author's claim was based on reading through the literatures as listed in the reference section. The completion of this study is important and necessary to the future construction of a 521-bit projective-domain Elliptic-curve public key infrastructure.

Definition of Terminology

The following terms are defined in the context of Elliptic-curve cryptography and this research. More detailed discussion of these terms can be found in standards (IEEE 982.1, 1988), *IEEE Standard Dictionary of Measures to Produce Reliable Software*, in IEEE 610.12, 2002), *Standard Glossary of Software Engineering Terminology,* in (ISBSG, 2006), *Glossary of Terms,* or in (IEEE 1363, 2000), *Standard Specifications for Public-Key Cryptography*.

Accuracy: Measurement that is closer to the actual.

Cost: Any measure, such as latency, of quantitative properties that has to be spent to obtain the result of the product $k(x, y)$.

Cost Index: A value that has been normalized from the cost value.

Homomorphism: Homomorphism allows mapping the numbers back into themselves. Homomorphism is a structure-preserving map between two algebraic structures of affine coordinates (x, y) and projective coordinates (X, Y) operable in groups, sub-groups, or fields.

Group: A group G is defined as a set, in which it is subsequently possible to define a binary operation that has an identity element, and has multiplicative inverses for each of its elements. The cryptology PEPMA works with large elements in the group; for this reason, the properties of the group are enormous and a complete understanding of the group is impossible. This led to a more practical approach in studying the properties of smaller groups by looking at a subset of a known group under a specific modulus (e.g. modulus m). This smaller group is known as cyclic subgroup inside a finite field F.

Field: An algebraic system consisting of a set S, two operations O1, O2 and their respective inverse operations, and two identity elements I1, I2, one for each operation.

K = (S, O1, O2, I1, I2)

S is a set of integers

O1 is the operation of addition. The inverse operation is subtraction.

O2 is the operation of multiplication. The inverse operation is defined below.

I1 is the identity element zero (0)

I2 is the identity element one (1)

Inverse: The word "*inverse*" is used in this context to indicate a numerical inversion of a polynomial with its presentation as a multi-digit number. Let *F* be forwarding functions of the variable *x*, and *x* is invertible if there exists a function *R* in domain *X* and range *Y*, with the following properties:

$$F(x) = x \quad \textit{iff} \quad R(\frac{1}{x}) = x$$

Let *r* be the inverse of *x*, then this congruent modulo must be true in domain *X* and range [0 ... *m*–1]

$$x \times r \equiv \ 1 \bmod m$$

If a multi-digit number, *MDN,* is invertible, then the inverse of *MDN* is unique; in other words, there can be at most one MDN^{-1} satisfying the inverse properties.

Performance Formula: a mathematical relationship or rule expressed in symbols used for calculating the performance of PEPMA. Performance formulas are components of KPI.

Key Performance Indicator: (KPI). It is an indicator which is used to determine how an evaluator will apply it against objectives. PKI has the ability to provide recommendation for course of action. For example, overall performance of PEPMA is a KPI.

Measurement: Measurement provides a single-point-in-space view of PEPMA specific, discrete factors. Measurement is generated by counting.

Metric: Statement of measurement. Metric is derived by comparison of predetermined baseline two or more measurements. Metric is generated by analysis. A metric can be absolute or a ratio. Thus metric can be of type "absolute metric" or "ratio metric". Metrics are components of formulas.

Reverse: The word "reverse" does not have the same meaning mathematically as "inverse." It is intended to indicate transformation functions that undo other transformation functions.

Precision: Measurement that is consistent for every reading.

Projectivity: A transformation within and between projective spaces.

Program Profiling: Investigation of PEPMA executing instructions**.**

Verification: The software engineering activities include testing, inspection, design analysis, and/or specification analysis to confirm that the performance formulas meet specifications levied on the design. Verification activities help produce high-quality performance formulas and metrics.

Point Doubling and Point Adding Definition:

By algebraic laws, a point-doubling of a point (x, y) on the curve $y^2 = x^3 + ax + b$ results in a second point (x_3, y_3) whose coordinates must also be on the curve. This result (x_3, y_3) is defined by two Cartesian coordinate equations in the Euclidian plane:

$$x_3 = \left(\frac{3x^2 + a}{2y}\right)^2 - 2x, \qquad y_3 = \left(\frac{3x^2 + a}{2y}\right) \times (x - x_3) - y$$

Similarly for point-adding, by algebraic laws, adding point (x_1, y_1) to point (x_2, y_2) on the curve $y^2 = x^3 + ax + b$ results in a third point (x_3, y_3) whose coordinates must also be on the curve. This result (x_3, y_3) is defined by two Cartesian coordinate equations in the Euclidian plane:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2, \qquad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) \times (x_1 - x_3) - y_1$$

Point-doubling of point p (x, y) is defined as adding the same point together

$$\text{Point-doubling} \triangleq p + p$$

Point-doubling is also equivalent to a multiplication of a scalar 2 and a point p

$$\text{Point-doubling} \triangleq 2 \times p\ (x, y).$$

Adding two different points p_1 (x_1, y_1) to p_2 (x_2, y_2) is not the same as point-doubling. Rather, point-adding is defined as adding two EC points having different coordinates:

$$\text{Point-adding} \triangleq p_1 + p_2$$

Mersenne prime: a prime of the form $2^p - 1$ where p is a prime. The NIST P-521 modulus m has a form of a Mersenne prime.

Mathematical Symbol

$\equiv$	congruency
$\triangleq$	equal by definition
$\mathbb{Z}$	integer number set
$\Leftrightarrow$	transformation
m	spatial modulus
N	digit count in a number, sometime referred to as limbs
$\lceil x \rceil$	rounds number to upper integer
$\lfloor x \rfloor$	rounds number to lower integer
mod	remainder calculation
= =	equality
=	assignment of value
[ ]	square bracket, digit index
K	an integer field containing elements of $\mathbb{Z}$
F	an infinite field
x, y	variables in Cartesian coordinates, or affine coordinates (lowercase italic)
X, Y, Z	variables in projective domain (uppercase non-italic)
$k(x,y)$	Elliptic-curve multiplication of scalar k and point p having affine coordinates (x, y)
$\approx$	approximately equality

Acronym

ARM	Advanced RISC Machine
CPU	Central Processing Unit
CISC	Common Instruction-set Computer
ECDH	Elliptic-curve Diffie-Hellman, a protocol to exchange private keys in public domain
EC	Elliptic curve
EPM	Elliptic-curve Point Multiplication
GF(p)	Galois prime field
MDN	Multi-digit Number, a big-number
NSS	Network Security Services, and open-source of cryptographic library
OpenSSL	Open Secured Socket Layer, and open-source of cryptographic library
PEPMA	Projective Elliptic-curve Point Multiplication Agent
PMS	Performance Measurement System
KPI	Key Performance Indicator
RSA	Ron Rivest, Adi Shamir and Leonard Adleman public-key encryption algorithm
RISC	Reduced Instruction-set Computer

Chapter Summary

The beginning part of the introduction briefly presented the problem to investigate. It is believed that the performance comparison between PEPMAs would be unknown based on coarse performance metrics M, S, and *clock*().

The performance comparison of PEPMA in a 64-bit x86 environment has three important, top-level contexts: (a) the structure of PEPMA, (b) associated environment of PEPMA, and (c) existing and expected methods for comparison and verification. The top two contexts were briefly presented in the preliminary section on PEPMA and in the section on how environmental factors would affect the performance of PEPMA.

To advance the investigation, two optimizing PEPMA codes, NSS and OpenSSL, will be selected for the empirical case study. These two open-sources, coupled with methods for comparison and verification, will help answer which metrics will objectively evaluate PEPMA's efficiency and ways to improve PEPMA’s efficiency based on the empirical comparison.

Chapter 2

Literature Review

The literature review is a collection of research papers, journals and reports that have been gathered as a basis for the evaluation and comparison of PEPMA in a 64-bit x86 run-time environment. The review is to locate a set of widely accepted principles in the area of concern. Based on this prior research, a common ground for the Performance Measurement System (PMS) of PEPMA can be characterized. Accordingly, the outcome of prior research and analysis will be used as a source of input to support the performance evaluation methods of PEPMA. Critical knowledge and substantive findings include: (a) performance evaluation using IEEE standards, (b) Elliptic-curve principles and their components and (c) the concept of point computation in projective geometry. These three important topics will point out specific formulations, theories, requirements, and analytical methodology to help evaluate PEPMA’s performance and support the comparison of performance between NSS and OpenSSL.

PMS principles indicated that when measuring PEPMA, various aspects of evaluation must be taken into account. Current "state of the art" knowledge includes up-to-date evaluation methods. Results and empirical data from these evaluators will facilitate an understanding of the structures and relationships among various measures of NSS and OpenSSL PEPMA.

The primary purpose of the arithmetic literature review, including a big-number representation, is to ascertain whether the proposed metrics can objectively evaluate PEPMA's efficiency and whether there are effective ways to improve PEPMA’s efficiency based on the empirical comparison. Subsequently, critical points of knowledge

about the Elliptic-curve field, projective geometry, and optimization efforts on low-level arithmetic will provide best practices for determining the optimum computationally efficient PEPMA under x86 64-bit platforms.

Performance Evaluation Standards

The purpose for reviewing the following standards and sub-components of the formulas is to assist in developing a formal evaluation methodology that will address questions from the research. It is important to derive accurate quantitative computational metrics available from a 64-bit executable environment and provided such measurements to a performance specialist.

One cannot reasonably evaluate performance accurately without first investigating the measurement principles noted in Shukri's paper that "software measurement science should use the same basic principles as physical measurement science, which requires a reference, measurement method, and an uncertainty statement" (Shukri et al., 1999, p. 3). While Shukri's measurement method refers to specific formulations recommended in the NIST standards, PEPMA's key performance measurement method relies on IEEE standards. Furthermore, while Shrukri requires the uncertainty parametric such that "the behavior conforming to the chosen reference, and options the reference permits" (p. 4) to be included in the measurement equations, the efficiency of PEPMA derives its uncertainty parametric though program profiling and simulation.

Both IEEE standards 982.1-1988 and 982.1-2005 provide some, but incomplete measurement references suitable for the Key Performance Indicators of PEPMA. However, efforts to tailor the efficiency measurements are necessary because no standards exist in this area. These IEEE standards also offer a recommendation for continual self-assessment and improvement of the software aspects of dependability. Within the revision released in 2005, IEEE 982.1 stated its boundary in the scope that "this standard specifies and classifies measures of the software aspects of dependability. It is an expansion of the scope of the existing standard; the revision includes the following aspects of dependability: reliability, availability, and maintainability of software. The applicability of this standard is any software system; in particular, it applies to mission-critical systems, where high reliability, availability, and maintainability are of utmost importance..." (p. 2). Under the limited capacity of these IEEE standards, the terminology and metrical formulations pertaining to the availability of critical systems are generally applicable to the Key Performance Indicators of PEPMA.

Specific definitions of primitives and formulations pertaining to software reliability are found in IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE 982.1 (1988), which is an older and original version of IEEE 982.1 (2005). Although this standard was published in the early microprocessor computing era of 1988, it was not revised until 2005. Regardless of the deficiencies in some areas, there are useful metrics to evaluate software reliability, which can be applied to ensure that PEPMA's top-level software module and some sub-modules exhibit accurate, consistent, repeatable, and predictable performance under a 64-bit environment.

In particular, IEEE 982.1 (1988) provides formulations of the following five metrics: Compliance Metric, Static Complexity, Weighted Information Flow Complexity, Module Maturity Index and Functional Metric. The methodology chapter will present applications of these formulas, along with the acquisition of sub-components of the formulas, known as primitives.

Despite the fact that the performance evaluation presented in the IEEE literature is incomplete, Herrmann (2007, p. 111) cross-referenced the software and security engineering metrics in her book, as she made an observation that, "Although not recognized as such, software engineering is also a first cousin of security engineering". She also noted that software engineering metrics defined by IEEE standards have been proven and "passed the accuracy, precision, validity, and correctness test" (Herrmann 2007, p. 120). Although Herrmann did not mention where one can find the origin of formulations for the Key Performance Indicators and other formulas as she presented, they were probably derived from the recommendations in standards IEEE 982.1 (1988) and IEEE 982.1 (2005). Another source that discusses these topics is available from Keyes (2005).

Efficiency Measurement

Efficiency measurement is one of most significant aspects beside other Key Performance Indicators in the processing of PEPMA in real-time. The efficiency will be measured with respect to its main objective, which is a minimization of computing costs in terms of reducing the number of CPU instructions. To date, there have not been significant standards available for evaluating the efficiency of Elliptic-curve point-multiplication in a projective domain. In an attempt to address this issue, general

discussion of this topic can be found scattered in Keyes's literature (Keyes et al., 2005) and many other research papers presented in the following review sections. Additionally, because there is not an absolute reference that PEPMA's efficiency measurement can be based on, the reference for measuring PEPMA's efficiency will be relative ─ meaning in between efficiencies of NSS and OpenSSL. Hence, the efficiency metrics are best if the following attributes are presented: they have ground truth, have a formal technical approach, are quantitative, are objective, are obtainable, are inexpensive to derive, are repeatable, and are verifiable. Certainly, the evaluation for efficiency might not be able to encompass all of those attributes. However, a few important ones – such as the empirical verification through program profiling and simulation – should be included. Taken from NSS and OpenSSL C source codes, Table 1 provides an incomplete list of similarities and differences between NSS and OpenSSL implementations that will potentially contribute to the point of reference for comparison and efficiency metrics.

Table 1. NSS and OpenSSL Similarity and Difference

NSS Unit of Analysis	**OpenSSL Unit of Analysis**	**Comment**	**Compared in between**
APT	APT	APT = Affine to Projective Transformation	Similar
4-bit windows and pre-comp EF	5-bit windows and pre-comp EF	EF = Exponentiation Function	Different
Point Doubling type 1, (Cohen et al., 1998)	Point Doubling type 2 (Brown et al., 2001)		Different
Point Adding type 1 (Brown et al., 2001)	Point Adding type 2 (Brown et al., 2001)		Different
PAT	PAT	PAT = Projective to Affine Transformation	Similar
32-bit numeric representation	58-bit numeric representation		Different

Elliptic-Curve Principles in PEPMA

One important element of PEPMA pertains to Elliptic-curve Cryptography (EC). There are many theories in this area since the discussion of EC began early in the Isaac Newton era. The discussions on this topic are included in several sources (Avanzi et al., 2006; Burton et al., 2006; Aoki et al., 2001; Brown et al., 2001). In this Elliptic-curve preliminary, principles closely related to helping the performance comparison of PEPMA, will be extracted and presented.

It has been shown that many Elliptic curves exist in a three-dimensional torus (Cohen et al., 2006, pp. 272-273), which is a donut-shaped object shown in Figure 9 below (Hankerson et al., 2004, p. 75-86).

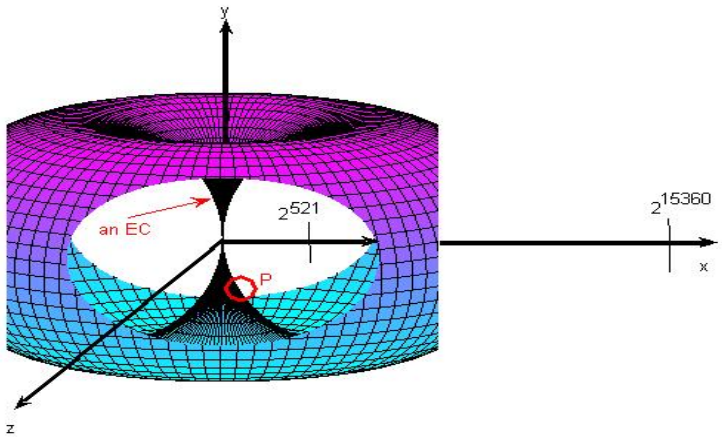

Figure 9. A 521-bit Elliptic-curve Point vs. 15,360-bit RSA Cryptographic Key

Of these curves, the 2D locus of points p on a Cartesian x-y plane must satisfy an algebraic cubic equation of the form $y^2 = dx^3 + cx^2 + ax + b$. In contrast with one-dimensional RSA cryptography, the distinction between a two-dimensional point p of an Elliptic-curve residing in a torus and a one-dimensional 15360-bit RSA cryptographic

key sitting along the *x*-axis is depicted in the Cartesian (*x*, *y*, *z*) coordinate system on Figure 9. Hankerson et al. (2004, pp. 15-19) and Certicom discusses this topic (Certicom Research, 2009; Certicom Research, 2004). Online information regarding key size is also posted at (RSA Key Size, 2013). Not all Elliptic curves are good for cryptography because they can be easily exploited or too difficult to manipulate in the forward direction. Readers are referred elsewhere for discussion on this topic (Bos et al., 2014). One particular curve P-521 per NIST recommendation has the cubic form:

$$y^2 = x^3 + ax + b$$

where the coefficients in a larger curve $y^2 = dx^3 + cx^2 + ax + b$ have been set to $d = 1$, $c = 0$, $a = -3$, and the constant b, known as the domain curve's parameters, will be selected at run-time. These coefficients, a and b, are usually stored in an X.509 certificate for public-key management (ITU-X509, 2014). This same curve, P-521, was used in NSS and OpenSSL implementation; but since b is variable, will its value change the latency of PEPMA at all? By setting $b = 0$, the NIST curve's appearance in (x, y) coordinates is depicted by the blue graphs below.

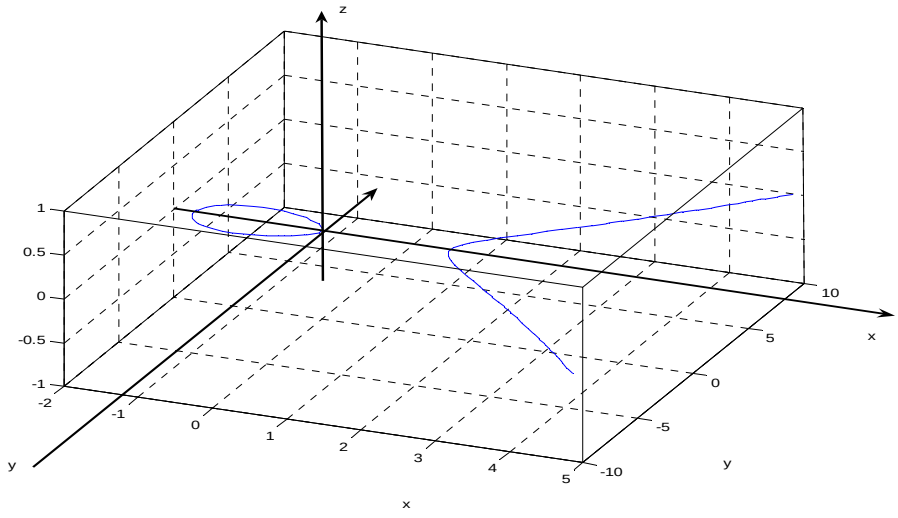

Figure 10. NIST EC in the Domain of Real Numbers in 3D

The solution associated with NIST Elliptic-curve in equation $y^2 = x^3 + ax + b$ can be performed in real, imaginary, binary, or prime-field numbers. It is natural to work with curves within an algebraic closure of real numbers. However, it has been shown that real, imaginary, and small-value numbers are used mainly for illustrating and understanding point addition, multiplication, squaring, and inversion, but it has no practical use for cryptology (Hankerson et al., 2004, pp. 80-82).

Since the NIST P-521 curve has a form $y^2 = x^3 + ax + b$, by setting $b = 0$, the EC now becomes an even simpler equation, $y^2 = x^3 - 3x + 0$. Definitions for a family of curves were established and published in NIST 186-2, or in FIPS PUB 186-4 (2013).

The equation $y^2 = x^3 - 3x + 0$ can be equivalently written as $y^2 = x\left(x^2 - 3\right)$. The three coordinates and 2D graph of this curve bounded in small real numbers are depicted in Figure 11 below. The red curve shows another infinite subfield when b = 3. This red curve will have different base points G(x, y) from the blue curve, although they are in the same family. By changing the domain parameter b and having different base points G(x, y), the illustration shows that calculating computational efficiency for the red curve might not be the same as for the blue curve.

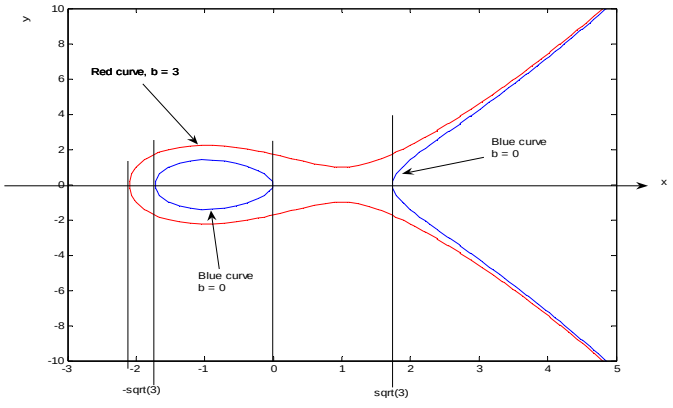

Figure 11. NIST EC in Small Numbers

When y equals 0, then $0 = x\left(x^2 - 3\right)$; this equality implies that there are, at most, three distinct roots for the curve. These three roots must all satisfy the equation $0 = x\left(x^2 - 3\right)$:

$$x = 0\,,\ x = +\sqrt{3}\ \text{, and}\ x = -\sqrt{3}$$

Thus, three distinct points, *p,* exist on the curve: $p(0, 0)$, $p(\sqrt{3}, 0)$ and $p(-\sqrt{3}, 0)$.

Based on algebraic laws, a point-doubling operation of a point $p = (x, y)$ on a P-521 EC will result in another point $p_3(x_3, y_3)$ that is defined by two Cartesian coordinate equations in the Euclidian plane (Hankerson et al., 2004, p. 80):

$$x_3 = \left(\frac{3x^2 - 3}{2y}\right)^2 - 2x\,, \qquad y_3 = \left(\frac{3x^2 - 3}{2y}\right) \times \left(x - x_3\right) - y$$

By substituting point $p(x = \sqrt{3}, y = 0)$ to (1), the product of scalar 2 and EC point p yields $k \times p = 2 \times p\ (\sqrt{3}, 0) = p_3(x_3 = \infty, y_3 = \infty)$.

In the process of calculating the parameters x_3 and y_3, one needs to take the inverse of $2y$, including when $y = 0$. This inversion operation will be expensive even with real numbers, more so if the arithmetic was done with 521 bits in a finite field. In NSS or OpenSSL, truthful performance evaluation must account for the condition where $p_3(x_3 = \infty, y_3 = \infty)$. It has been shown that both point-doubling and point-adding functions must handle this peculiar mathematic condition known as the processing of a point at infinity (Cohen et al., 2006, pp. 268-271).

In an Elliptic-curve crypto system, the key length denotes a number of binary bits. PEPMA will manufacture cryptographic keys as a coordinate (x, y) of a point p in a finite-field 2-dimensional space. Thus, the length for each big-number x or y could range from 128 bits to 576 bits, depending on the security strength requirements. It has been shown that 521-bit key works well mathematically under a prime field (See Appendix J)

or (Cohen et al., 2006, p. 182); thus, it will be the chosen crypto key length in this research. NIST Special Publication 800-56A also discussed this topic.

Concept of Point Computation in Projective Domain

The concept of point computation in a Projective domain will explain why the existence of coordinate Z and why the first projective coordinate to enter the exponentiation computational loop has a value of (X = *x*, Y = *y*, Z = 1). The literature from (Rovenski, 2006; Ryabko et al., 2005, p. 98; Veblan et al., 1906) provided discussion of this topic.

Additionally, the concept will help answer many other questions about efficient computation in projective geometry. Salomon (2006) briefly described this conception in a 3x3 matrix representation (p. 13).

Boston and Darnall in literature have researched this mathematically intense topic (Koc, 2009). They noted that although an Elliptic curve having one genus (one doughnut-hole) is a subset of Hyperelliptic curves, formulations derived for Hyperelliptic curves can be used among different families of curves, such as the formulation for counting points on a Jacobian curve JAC(C). They further noted that to compute *k*P for some element P in Jac(C) and the order $n \in Z$ using the standard double-and-add method, one would be forced to expend a computational cost of $O(\log_2(n))$ inversions[4]. Boston and Darnall also showed that the high cost of inversions in an affine coordinate is usually valid for software. The final evaluation would signify a higher-performance improvement if one performs the comparison between a non-weighed projective transformation system

[4] When using Boston and Darnall's formulation, $\log_2(n)$ cost index in PEPMA equals exactly 521 inversions per double-and-add method.

and a system that uses a weighed projective transformation. Naturally, both NSS and OpenSSL PEPMA 2013 releases used the weighed projective transformation to obtain higher efficiency.

Boston and Darnall indicated that by introducing another variable Z, it is possible to delay performing inversions until the last step of the algorithm. They also noted that for Elliptic curves, this extra coordinate Z is equivalent to storing the point in projective coordinates. This is not the case for higher genus curves greater than 1; however, they still called these coordinates projective because of the similarity to Elliptic curves. Their notion helps clarify the concept of point computation in projective geometry. In turn, it distinguishes between different approaches implementing point-doubling or point-adding functions.

Joye also included the performance comparison between Jacobian and Chudnovsky coordinates (Joye, 2008). His idea of saving one Multiplication, 1M, and one squaring, 1S was achieved by using two more new coordinates, E and F, additionally with the Jacobian representation of points. The Chudnovsky presentation of a point P then becomes P(X : Y : Z : E : F). Neither NSS PEPMA nor OpenSSL PEPMA uses point presentation in the form of Chudnowsky, but uses three projective coordinates in a point P(X : Y : Z) described as *Projective-3* per Bernstein and Lange (Bernstein & Lange, 2007).

In 2007, Bernstein joined efforts with Lange, and together they published a paper titled "Analysis and Optimization of Elliptic-curve Single-scalar Multiplication." Their work was supported in part by both the National Science Foundation and the European Commission through the IST Programme (Bernstein & Lange, 2007). In their research,

they intended to present greater precision on how many field multiplications were required for the computation of *k*P.

To further elaborate the reasoning of other researchers in the projective field, Ryabko et al. (2005) and Salomon (2006) explain the same idea in their work. Other mathematicians (Cohen et al., 2006; Case, 2006) studying the projective geometry first examined the computation of an Elliptic-Curve point by drawing a point *p*, which has Euclidian 2D coordinates (*x*, *y*). This point *p*, shown in the previous graph as $p(\sqrt{3}, 0)$, represents a point on an Elliptic curve E. For simplicity, coordinate *y* is set to zero. Point *p* is situated on a flat surface π and in a Cartesian "*x-y*" coordinate system as shown in Figure 12 below. General discussion of projective geometry can be found in (Cohen et al., 2006, p. 46).

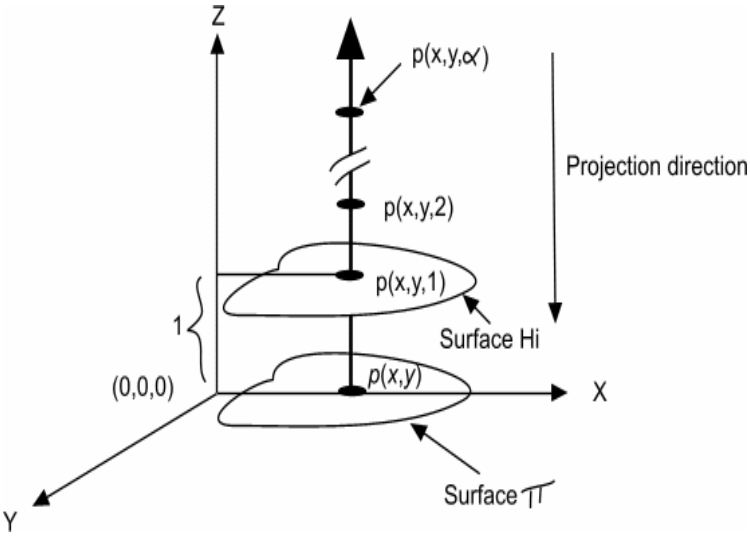

Figure 12. Transforming an Elliptic Point onto Projective Geometry

Point *p* (italic letter) is then lifted upward one unit in the *z* direction. Point *p* now becomes another point p (non-italic letter) that has an additional *z* coordinate equal to 1. The vertical movement engages point *p* (italic letter) to enclose an Euclidian distance vector equal to (*x*, *y*, 1) but this point is still in the Cartesian coordinate system.

However, from PEPMA's computational perspective, coordinate z =1 has no effect on the result. Thus, both points *p* (italic letter) and point p (non-italic letter) are the same point. In referencing the points located on an axis parallel to the z axis shown in Figure 12, any scalar value where $z = \alpha$, point *p* and its many other point p's located vertically above or below point *p* are all identical. Ryabko et al. (2005, pp. 99-101) declared such equivalency as follows:

$$(x , y) \triangleq (x , y , 1)$$

In technical terms, all points p(s) having coordinates (x, y, α) are "*homogeneous*" with respect to point *p* for the reason that they all represent the same point *p* that exists in Euclidean space (Bennett, 1995). Because of the *homogeneity* of point p (non-italic letter), the flat surface "Hi" in Figure 12 can also be thought of as a projective plane submerged in a homogenous coordinate system (Greenberg, 1995). The transition from affine coordinates (x, y) to projective geometry containing the first point P(x, y, 1) and immediately back to affine (x, y) is shown in the figure below.

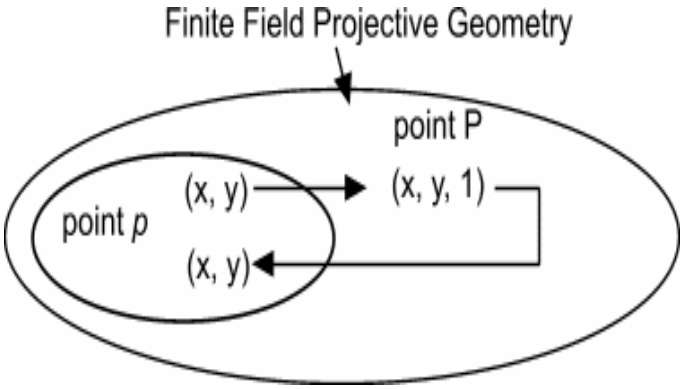

Figure 13. First Movement of EC Point Onto PG and Back

The original Euclidian point *p* can now be correctly derived from its homogenous coordinate to the Cartesian coordinate by "*mapping*," or making a projection of *p* onto plane π on the projection axis parallel to the Cartesian axis z (Greenberg, 1995). This

calculation is valid since a chosen working finite field *F* within Euclidean geometry is completely contained within a finite field *K* of Projective Geometry (Rovenski, 2006).

Because the transformation of (x, y) ↔ (*x*, *y*, 1) has already brought (*x*, *y*) into a homogenous coordinate system, for distinction of notation, one can denote point p with upper-case letters instead:

$$p(x, y, 1) \triangleq P(X, Y, 1)$$

To further elaborate the analysis of the projective field, let another point Q with its homogenous coordinates (X_3, Y_3, Z_3) shown in Figure 14 be a point which its values are the result of the projectivity of point P along axis vector V (Ryabko et al., 2005, pp. 99-101). This vector, V, passes through the Cartesian original point (0, 0, 0).

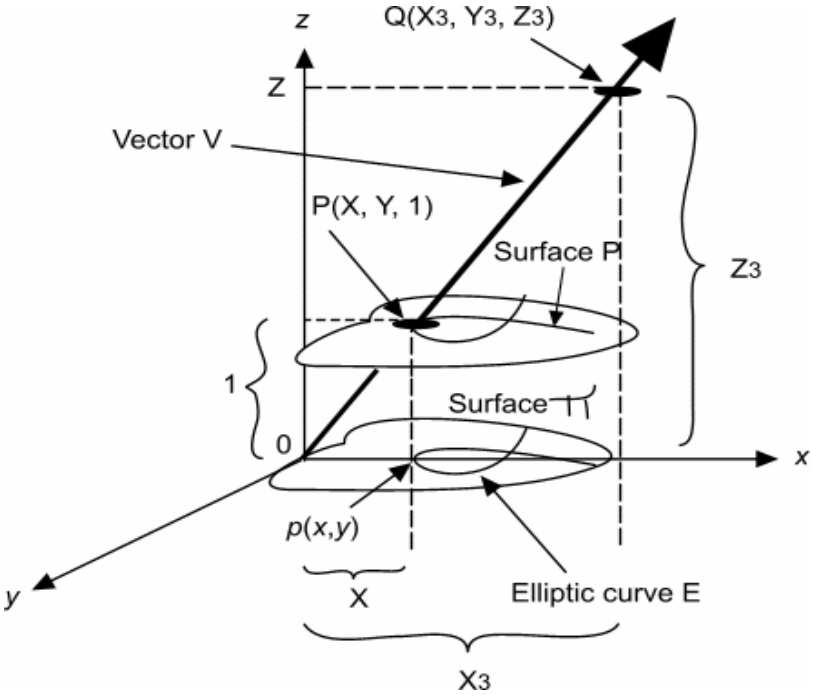

Figure 14. Projectivity of Elliptic Points

Due to the principle of *similar* triangles, the relationship between the two coordinates P and Q is given by

$$\frac{X}{X_3} = \frac{1}{Z_3} \qquad \text{and} \qquad \frac{Y}{Y_3} = \frac{1}{Z_3}$$

Under the working finite field *F* and the chosen modulus *m*, a 521-bit NIST Mersenne prime (Solinas, 1999) for example, and by the congruent relationship of *similar* triangles, homogenous coordinate X_3 can be derived such that

$$\frac{X}{X_3} = \frac{1}{Z_3} \rightarrow \quad X_3 \equiv \left(X \times Z_3\right) \mod m \qquad (2.1)$$

and the homogenous coordinate Y_3 is given by

$$\frac{Y}{Y_3} = \frac{1}{Z_3} \rightarrow \quad Y_3 \equiv \left(Y \times Z_3\right) \mod m \qquad (2.2)$$

In applying expressions (2.1) and (2.2), point Q will be correctly projected back to point P, provided an inversion of Z_3 modulo *m* exists. Once point P has been recovered by reversing the projectivity of point Q, one can derive the Cartesian coordinates (*x*, *y*) right after a projective transformation reverse (Cantor, 1987). Figure 15 illustrates a complete computation loop of an Elliptic-Curve point *p* in the projective domain. The term PT denotes any Projective Transformation, weighted or non-weighted.

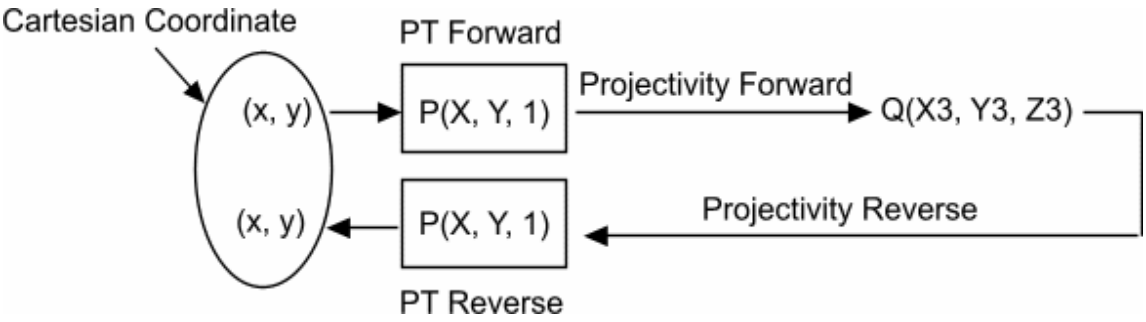

Figure 15. PT and Projectivity of EC Points

Assuming that point $p(x, y)$ will be transformed into a homogenous coordinate by the chosen weighted relationship

$$X = x \times Z^2 \qquad Y = y \times Z^3 \qquad Z = z \tag{2.3}$$

Accordingly, a constant vector $\alpha(1,1,1)$ in Cartesian has three homogenous coordinates

$$\Lambda\ (Z^2,\ Z^3,\ Z) \tag{2.4}$$

Using the principle of homogeneity in projective geometry, one can multiply coordinates of Q by a homogenous vector without changing its perspectives (Blake, 2001; Cantor, 1987).

$$Q(X, Y, Z) \triangleq \Lambda \times Q(X, Y, Z) \tag{2.5}$$

Due to the inversions required in expressions (2.1) and (2.2), the result of a point-doubling or point-adding in a projective-domain finite-field yields three coordinates with two inversions in place

$$Q\left(\quad \left(\frac{X_3}{Z_X} \bmod \quad m \right) \quad , \quad \left(\frac{Y_3}{Z_Y} \bmod \quad m \right) \quad , \quad Z? \quad \right) \tag{2.6}$$

where X_3, Y_3, Z_X, and Z_Y are the labels of the result of point-doubling or point-adding done in projective domain. Point Q has a "Z?" at *z*-coordinate because its value will be determined later in the computation process. Multiplying point Q with the homogenous vector Λ will yield the same point. The challenge for NSS and OpenSSL developers implementing PEPMA is to find a commonality between the terms Z, Z_X and Z_Y, such that inversions in the *x* and *y*-coordinates will be eliminated. Ryabko et al. (2005) described and derived this common denominator using substitution of variables and reduction of mathematical terms (p. 99). Assuming such a commonality is found,

$$Z_X = Z^2 \qquad \text{and} \qquad Z_Y = Z^3 \tag{2.7}$$

then the computations in expression (2.6) require no inversions, but the term Z must be carried onto the next computation of Z (EFD, 2001). The homogenous vector Q is just

$$Q(X_3, \quad Y_3, \quad Z) \tag{2.8}$$

Expression (2.8) concludes that point Q, the result of point-doubling or point-adding, can be manipulated in a homogenous coordinate without any modulo inversions.

In summary, findings from mathematicians in projective geometry have indicated that a projective space of an Elliptic curve can be formed by mapping vector spaces along a line through origin O. Additionally, projective geometry where the Elliptic curve reshaped is a non-metrical form of geometry. This means that coordinates associated with projectivity are no longer based on the concept of Euclidian distance. However, when the projective space of an Elliptic curve is projected back onto the Euclidean plane, the original coordinates presented in finite-field big-numbers will be restored. The homogeneous characteristic in projective geometry makes the exclusion of mathematical inversions possible.

Point at Infinity

While observing stars in the sky hundred of years before Poncelet, an important concept regarding a point at infinity appeared to the German mathematician and astronomer Kepler (1571). Today, in the principle of Elliptic-curve point computation, a point O at infinity must exist, be presentable, and be calculable. Thus, PEPMA's performance measurements will be affected by how a point-at-infinity is presented and processed. The point at infinity principle will be of assistance in selecting test vectors for PEPMA composed of order *n* of subfield and base point G(*x*, *y*) from domain parameters.

Computation in Mixed Coordinate

Cohen, Miyaji, and Ono presented an application of mixed coordinates, a combination of affine and projective computation toward Elliptic curve exponentiation in an article titled "*Efficient Elliptic Curve Exponentiation using Mixed Coordinates.*" Their research shaped the mathematical foundation for computations in a projective domain with mixed coordinates (Cohen et al., 1998). Both NSS PEPMA and OpenSSL PEPMA use the mixed coordinates approach to save costs during pre-computation. Thus, an accurate performance evaluation should account for this “mixed coordinate” condition as well.

PEPMA Domain Parameters

Processing PEPMA requires additional parameters associated with the characteristics of the curve. These domain parameters are chosen based on certain security criteria and performance levels. They are also based on the possible attacks that can be instigated on an Elliptic curve cryptosystem. For this reason, the ANSI X9.62, NIST 186-2 and IEEE standards provide the acceptable global parameters for all fifteen Elliptic curves.

Table 2. NIST P-521 Domain Parameters (FIPS PUB 186-4, 2013, p. 16)

Description	**Letter**	**Value (521 bits)**
Field size	*m*	000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
Coefficient for the Elliptic curve equation (521 bits)	*a*	−3 (decimal) 000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFC
Coefficient for the Elliptic curve equation (521 bits)	*b*	00000051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00
Order of the curve (521 bits)	*n*	000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783

		BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409
Cofactor *h*		1

Documentation in NIST 186-2 identified fifteen sets of parameters: five for prime fields, five for binary fields, and five for Koblitz curves. These parameters have been chosen for fast reduction with their respective modulo. A question here is whether NSS and OpenSSL PEPMA have applied the fast NIST modulo reduction using the Mersenne prime modulus (Solinas, 1999), since this mathematical optimization is a major contributor to the efficiency of PEPMA.

One of the other not so obvious parameters from NIST was the cofactor *h*: product of the cofactor and the order of the curve equals to the number of points on the chosen Elliptic curve: $h \times n = \# E(GF(p))$. For a more detailed discussion of cofactor *h*, readers are referred to (FIPS PUB 186-4, 2013, p. 87) or (NIST, 2010).

NIST curves utilize a cofactor term with a value of 1. However, determining the order *n* for an NIST curve requires a way to count the number of points available from the curve. These principles and findings will help select test vectors for PEPMA composed of order *n* of subfield and base point G(*x*, *y*) from domain parameters.

Research in Numeric Presentation and Computation

Integer computation for PEPMA requires processing multi-digit multiplications, divisions, and inversions (NIST, 2010). If one views PEPMA as part of Digital Signal Processing (DSP) at the microscopic level, the EPM operation begins with a low-level multiplication of two polynomial signals.

Various numeric theories (Cohn, 1962) and DSP literatures (Li, 2008; Lathi, 1998) have described the problem of N-digit polynomial multiplication (Bi et al., 2004);

however, from a fundamental aspect, the multiplication of two polynomials is equivalent to the problem of convolving two sequences of N-point signals.

If one denotes both n and k as non-negative integers for indexing into an N-digit number, all multiplications and summations required in (2.9) can be done entirely in spatial domain. However, in light of reducing computational costs, an existing problem is computing the result $r[n]$ with a multiplication algorithm that has the least multiplications and additions among all algorithms that compute $r[n]$.

Let x be the polynomial multiplicand and y be the polynomial multiplier, the multiplication resulting in another polynomial r is obtainable by convolving x and y

$$r[n] = \sum_{k=0}^{N-1} x[k]y[n-k] = x[n] \otimes y[n] \qquad (2.9)$$

Computational efficiency of PEPMA begins with a proficient representation of a Multi-Digit Number (*MDN*). The following information sets up essential terminology for a discrete polynomial representation of a multi-digit number and prepares its usage for low-level arithmetic calls from PEPMA.

There are several ways to represent an *MDN* contained in a finite field K (Koc, 2009; Saldamli, 2009). However, two types of presentations are popular, the prime field F_p and the exponential prime field F_p^s.

If prime p is set to 2, then the exponential prime field F_p^s becomes F_2^s, an exponential binary field. Moreover, if the *MDN* is implemented using a two-bit field F_2, then NSS or OpenSSL PEPMA can represent an S-bit multi-digit number, contained in an exponential binary field, implemented over a two-bit field b as a finite discrete polynomial, along with a *sign* indicator.

$$MDN = (sign)\left[b_{S-1}(2^{S-1}) + b_{S-2}(2^{S-2}) + ... + b_1(2^1) + b_0(2^0) \right] \qquad (2.10)$$

For a more detailed discussion of numeric representation, readers are referred to (Cohen et al., 2006, p. 169) or (FIPS PUB 186-4, 2013, p. 88). Although expression (2.10) can provide a workable representation of *MDN* as a bit vector, the representation is not yet efficient for performing arithmetic between *MDN*s. Instead, almost all 64-bit computational units currently available in general processors provide signed arithmetic operations with arithmetic word lengths set to 32 (half-digit in 64-bit system), 58, 64, or 65 bits (64 bits plus hardware carry bit).

If one denotes arithmetic word lengths to be *L*, then the absolute minimum number of digits *N* required in an *S*-bit *MDN* is a ceiling function of *s* and *L*

$$N = \left\lceil \frac{S}{L} \right\rceil \qquad (2.11)$$

and the representation of an *MDN* as a signed digit vector is given by

$$\mathrm{MDN} = (sign)\left[\sum_{n=0}^{N-1} d_n(2^{nL}) \right] \qquad (2.12)$$

where digit *d* located at index *n* of an *MDN* is a weighed sum of the bit vector

$$d_{\mathrm{n}} = b_{nL+L-1}(2^{L-1}) + b_{nL+L-2}(2^{L-2}) + ... + b_{nL+1}(2^1) + b_{nL+0}(2^0) \qquad (2.13)$$

$$n = \{0, 1, 2, \ldots, N-2\}$$

Readers are referred to (Cohen et al., 2006, p. 171) for a general discussion of internal representation of a single-precision number (digit *d*). The summation in (2.12) is valid only for the lower *N*–2 digits, digit 0 to digit *N*–2. The most significant digit (MSD), located at index *N*–1, is different because the computation in Elliptic-Curve Point Multiplication (EPM) might not require all available bits in *N* digits. Upper zero bits in

NSS and OpenSSL architecture are used for arithmetic overflow. How upper zero bits are used will significantly affect efficiency.

For example, an EPM modulus is an *MDN* with S = 521 bits, but there are 544 bits available in seventeen 32-bit digits. This setting always creates 23 zeros unused in the most significant bits (MSB) in the digit N–1. The MSD of an *MDN* is a polynomial of degree L, but it has a different form if S is smaller than the product $N \times L$:

$$d_{N-1} = 0... + b_{S-1}(2^{L-1}) + b_{S-2}(2^{L-2}) + ... + b_{(N-1)L+1}(2^0) \qquad (2.14)$$

For a discussion of multi-precision number, readers are referred to (Cohen et al., 2006, p. 170). The representation of a Multi-Digit Number (*MDN*) in a computing platform's memory is illustrated in Figure 16 below:

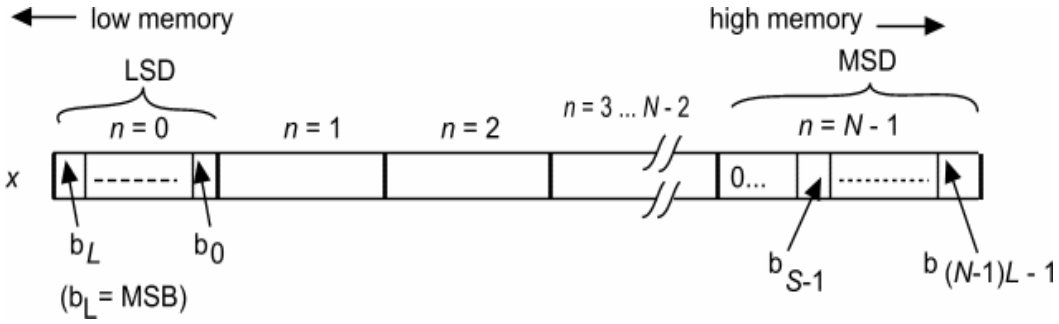

Figure 16. Representation of a Spatial Multi-Digit Number

Several particular cryptographic service routines in NSS and OpenSSL, such as finding an inverse of an *MDN* with an extended Euclidian algorithm, require a sign flag (*sign*) for each *MDN*, allowing the arithmetic operations to work properly. A single bit in F_2 will be adequate to indicate a plus or a minus sign of an *MDN*. Computations in NSS and OpenSSL PEPMA use a complete *MDN* with their representation shown in Figure 16.

Modulo Reduction

Methods for performing modulo reduction after multiplying two polynomials in spatial domain can be carried on with standard integer division (Maeder, 1996). The modulo reduction can also be carried on using NIST reduction method (Hankerson et al., 2004, p. 44; NIST, 2010). One can also choose an optimized division similar to Knuth's division technique (Knuth, 1985). With the selection of Knuth's algorithm, division in spatial domain becomes highly efficient in that the quotient and remainder converge to correct values fast; this quick convergence is possible because the error can be reduced quadratically on each iteration. Setting a modulo reduction method other than the NIST recommended way might change the efficiency of PEPMA. In this research, optimization of modulo reduction algorithm other than NIST's approach will not be considered.

Inversion

Both NSS and OpenSSL PEPMA calculate the inversion of a number in spatial domain using an extended Euclidean algorithm (Hankerson, 2004, p. 39). Given a number *n*, its multiplicative inverse *i*, and the modulus *m*, then the relationship of $i \times n \equiv 1 (\mathrm{mod}\, m)$ holds true between them.

Since *m* divides the term (*in* – 1), *m* is also a divisor of (*in* – 1). The relationship between *i*, *n* and *m* can be rewritten as $(i \times n) - 1 = bm$, where *b* is any non-zero integer. Rearranging the terms of the equation equivalently produces:

$$(i \times n) - (b \times m) = 1$$

Given *n* and *m*, the extended Euclidean algorithm discovers three unknowns *i*, *c* and *g* corresponding to the equation $in + cm = g$. Thus, if $g = 1$, then *i* is the inverse of *n*.

Although the extended Euclidean function can calculate cofactors *i*, *c* and gcd(*i*, *m*) quicker than Lagrange's exponential method, it still can be further optimized to provide additional efficiency for larger multi-digit numbers.

Inversion in Spatial Domain

Input:	*n* to inverse, modulo *m*
Output:	*i*, the multiplicative modulo inverse of *n*
Processing Unit:	Euclidean Algorithmic
Processing Cost:	$O([\log_2(m)]^2)$

Prior Research in Evaluating PEPMA

A comprehensive review on Elliptic curve cryptography for embedded systems has been provided (Afreen et al., 2011). This document graphically shows various methods of scalar point-multiplication *kp*. Figure 5 in the document describes the separation of *kp* into three levels of abstraction: (a) upper level for protocol such as ECDH, (b) middle level for Elliptic-curve Point-multiplication (EPM), and (c) low level for core arithmetic such as addition, subtraction, and multiplication. At the middle level, Afreen illustrated two different implementation methods: a projective coordinate standard projective algorithm and the Jacobian based approach. In its context, the standard projective implementation refers to Elliptic-curve arithmetic operations with orientation to projective geometry, while the Jacobian implementation refers to weighted Elliptic-curve arithmetic operations in the entire range of Jacobian space. Note that the comparison between NSS and OpenSSL of two EPMs in projective domain was limited to a specific weighted Jacobian computing space (Z^2 and Z^3). Their paper was recently published in the International Journal of Computer Science & Information Technology (IJCSIT) in 2011.

In the Journal of Computer article published in January 2010, the performance of Elliptic curves in projective coordinates with parallel computing in GF(p) was evaluated (Somani, 2010). Somani noted that projective-coordinate systems are used to eliminate the need for performing inversions. He found and recorded several projective-coordinate systems that had been proposed before his time. He noted that similar research on computation in projective-coordinate systems is recorded in Bernstein (2007). Somani describes how a homogeneous coordinate can be viewed as an Elliptic curve point *p* that takes the form $(x, y) = (X/Z, Y/Z)$. For the Jacobian coordinate system, point P takes the form $(x, y) = (X/Z^2, Y/Z^3)$. Without presenting any concrete proof, Somani provided the formula for point adding and point doubling in a projective-coordinate system.

Several critical articles for Elliptic-curve computations were collected (Cetin Kaya Koc 2009). Chapter 8, "Elliptic and Hyperelliptic Curve Cryptography," written by Nigel Boston and Matthew Darnall, also provides an introduction to the topic of elliptic and hyperelliptic curves.

An article, published in the Arithmetic of Finite Fields of Lecture Notes in Computer Science, described several strategies to speed up the arithmetic of Point-multiplication on Elliptic-curve using right-to-left and left-to-right methods (Joye, 2008). Both NSS and OpenSSL PEPMA use a right-to-left algorithm. In the point addition of section 2.2, Joye calculated the costs of adding two different Elliptic-curve points in the weighted projective coordinate system to be 12 multiplications (12M) and 4 squarings (4S).

Explicit Formulation

Explicit formulation offers specific formulas for calculating point-doubling and point-adding. Derivations and proofs for these formulas require mathematical intensive and tedious efforts. However, accessing the context of explicit-formula might be helpful for optimizing the point-doubling or point-adding function. For completeness, the derivation of explicit formulas will be listed in the report (Brown et al., 2001).

Bernstein and Lange introduced their Explicit-Formulas Database (EFD). It is a web-based collection of explicit formulas for elliptic-curve cryptology. Additionally, the EFD website has posted several useful formulas for other coordinate systems, such as Edward's curves. Bernstein and Lange designed the transformation as *Projective-3* and posted its cost for computing to be 12M+ 2S. Compared to a standard method, adding two points in a *Projective-3* coordinate system totally eliminates the inversion and, at the same time, increase multiplications from 1 to 12 and squarings from 1 to 2.

For doubling a point in a *Projective-3* coordinate, Bernstein and Lange posted the cost for computing to be 7M+ 3S. Compared to the standard method, doubling a point eliminates the inversion altogether but also increases the number of multiplications from 1 to 7 and the number of squarings from 1 to 3.

From a collection of each cost from the algorithm 3.21 for point doubling of curve $y^2 = x^3 - 3x + b$ in Jacobian coordinates[5], estimated costs can be accumulated as follows:

$T1 \leftarrow Z_1^2$	1 S
$T2 \leftarrow A = 3 \times (X_1 - Z_1^2) \times (X_1 + Z_1^2)$	2 M
$Y_3 \leftarrow B = 2 \times Y_1$	1 M

[5] This algorithm 3.21 is available from Vanstone's literature

$Y_3 \leftarrow B = 2 \times Y_1$	1 M
$Z_3 \leftarrow B \times Z_1$	1 M
$Y_3 \leftarrow C = B^2$	1 S
$T_3 \leftarrow D = C \times X_1$	1 M
$Y_3 \leftarrow C^2$	1 S
$Y_3 \leftarrow A^2$	1 S
$T_1 \leftarrow 2 \times D$	1 M
$T_1 \leftarrow (D - X_3) \times A$	1 M
$Y_3 \leftarrow (D - X_3) \times A - C^2 / 2$	1 M

The total cost turns out to be 9M+3S, which supports Bernstein and Lange's record of 7M+ 3S.

To evaluate PEPMA coarsely, researchers calculated the arithmetic costs in projective coordinates of a specific point-adding and point-doubling method and summarized them up to a total expenditure of mathematical operations for the scalar point-multiplication *kp*. In certain findings, total arithmetic expenditures to compute *kp* were 3668M + 3668S (Cohen et al., 1998; Brown et al., 2001); in Bernstein's findings, the total arithmetic expenditure was 2983M + 3275S (EFD_Double, 2001; EFD_Add, 2007). From these explicit expenditures, two metrics multiplications (M) and squarings (S) were the main coefficients of the cost equation to measure the performance of elliptic-curve point-multiplication *k*P residing in projective domain.

Theoretically, Bernstein's approach should be slightly faster than Cohen/Brown's method. However, in a 64-bit x86 run-time environment, timing costs do not correlate well to either Cohen/Brown's or Bernstein's total expenditures. In other words, the metrics M and S alone cannot provide truthful performance between two elliptic-curve scalar multiplications. Coarse-performance metrics can be used to validate the correctness of formulations.

Cohen and Frey collected a variety of articles belonging to the computations and optimizations of an Elliptic-curve (2006). One article written by Christophe Doche and Tanja Lange describes the arithmetic of elliptic curves. In section 13.2 (Choice of the coordinates), Doche and Tanja presented computations in an affine coordinate, computations with projective coordinates, and computations using mixed coordinates. This topic has been shown in (Cohen, 1998). The computation of EPM in a mixed coordinate was a new suggestion at that time. Both NSS and OpenSSL can activate the computation in non-mixed and mixed coordinates.

In 2004, Aigner, Bock, Hutter, and Wolkerstorfer from Infineon Technologies took a different approach toward the application of the *kp* process for computing EPM (Aigner et al., 2004). They applied EPM using an affine coordinate to a low-cost ECC coprocessor for smartcards. This custom-made, hardware-based co-processor runs a specific function to produce an Elliptic Curve Digital Signature (ECDSA) in a GF(2^m) field. Table 3 of their paper lists the performance for a 191-bit ECDSA algorithm. Table 3 is duplicated here for investigation (Aigner et al., 2004, p. 117). NIST has approved the use of test vectors for Elliptic Curve Digital Signature Algorithm as specified in ANSI X9.62 (ANSI, 2005) to informally verify the implementation.

Table 3. Performance of EPM in Hardware

	Operation clock cycle
Scalar Multiplication	341,430
30% overhead	102,429
GF(p) inversion	24,310
5% overhead	1,216
Total	469,385

In particular, Aigner et al. noted that having a fast GF(2^m) inversion makes it possible to use affine coordinates instead of projective coordinates for an elliptic-curve scalar point operation. This fast inversion is shown in the table above with 24,310 out of 469,385 clock cycles. Their paper marked a milestone in showing the best approaches for performing a *kp* function in hardware. Their findings reinforce Joye's theory about using mixed coordinates in computation to improve efficiency. NSS and Open SSL PEPMA switched this feature on and off under the user's command.

In 2000, the Microprocessor and Microcomputer Standards Committee of the IEEE Computer Society approved an IEEE Standard Specification for Public-Key Cryptography. This standard, (IEEE 1363, 2000), specifies common public-key cryptographic techniques, including mathematical primitives for deriving private keys, public-key encryption, digital signatures, and cryptographic schemes based on those primitives. It also specifies related cryptographic parameters, public keys, and private keys. The purpose of this standard is to provide a reference for a variety of calculating techniques from which applications may select.

In section A.10.5, projective elliptic addition (prime case), IEEE 1363-2000 defines the projective formulation for point adding on the curve $y^2 = x^3 + ax + b$ modulo m. The algorithm will consume ten field multiplications (10M) and five temporary variables.
In section A.10.4, page 124, projective elliptic doubling (prime case), (IEEE 1363, 2000) defines the projective formulation for point doubling on the same curve. The algorithm will consume sixteen field multiplications (16M) and seven temporary variables.

This IEEE 1363 marked a significant developmental point where the industry tried to standardize common computations, including computations in affine and computations

in projective coordinates. The findings can also direct better usage of temporary variables in point-doubling and point-adding to improve performance evaluation.

Chapter Summary

The literature review of PEPMA in a 64-bit x86 environment has two important, top-level contexts: (a) prior research on the structure of PEPMA and (b) existing methods for comparison and verification. Prior research on the structure of PEPMA was presented at the beginning of the literature review. This section seeks research pertaining to principles, findings, analysis in IEEE standards for Key Performance Indicators, Elliptic-curve principles, the concept of computation in projective coordinates, and big-number arithmetic representation. These four principles will answer which metrics can objectively evaluate PEPMA's efficiency.

To answer which metrics can be used for comparison and to subsequently provide ways to improve PEMA, this research will depend on kin topics: basic arithmetic service routines and modulo reduction. The main purpose of this literature review section is to ascertain whether the proposed metrics can truthfully evaluate PEPMA's efficiency and whether there are effective ways to improve PEPMA’s efficiency based on the empirical comparison.

Chapter 3

Methodology

Overview

Given a mixture of projective transformations, diversity of underlying arithmetic algorithms, and different computing platform architectures, the goal of this research is to provide suggestions to improve the projective Elliptic-curve point-multiplication agent. The objective is accomplished by dynamically or statically selecting higher-performance arithmetic approaches based on quantitative computational metrics. To fulfill the ultimate goal and to answer the particular research question of which metrics can truthfully evaluate PEPMA's efficiency, construction of a specific performance measurement system for PEPMA is necessary. The path to successfully derive Key Performance Indicators is illustrated in Figure 17. For a general discussion of Key Performance Indicators, readers are referred to the standard ISO/IEC 15939 (2001).

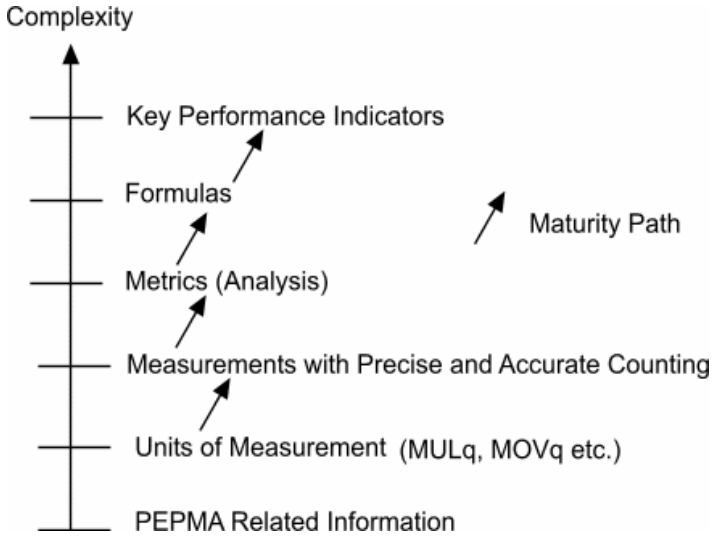

Figure 17. PEPMA Performance Measurement System

The PEPMA Performance Measurement System (PMS) can be briefly defined as a set of accurate, precise and quantifiable metrics applied to cost equations. The accuracy and precision of these metrics are derived from an analysis of the accurate and precise counting of the measurement units (MULq, MOVq etc.) Thus, the development of PMS will start at the bottom, defining the measurement units, and ladder up toward KPI through the maturity path. A general definition of PMS pertaining to the measurement and rating of performance of computer-based software systems can be found in (IEC-14756, 1999).

To date, there have not been significant standards available for developing the performance of Elliptic-curve point-multiplication in a projective domain. Accordingly, the construction of PMS will have to level on tailored models as described in software engineering (Herrmann, 2007; Fenton, 1996) and other comparable publications (Keyes, 2005). There have been three other comparable publications in the field of security and privacy metrics that can be applied to the construction of PMS: NIST SP 800-55 (NIST, 2008), and NIST SP 800-80 (NIST, 2003). In 2003, the National Institute of Standards and Technology released a Special Publication 800-80 titled "Guide for Developing Performance Metrics for Information Security." These publications in the field of security and privacy metrics can be applied to the construction of PMS as well.

The research on the construction of a performance measurement system for PEPMA is quantitative and primary[6]. Following suggestions from Pare (2004) and Gillman (2003), this research used an empirical case study with the following components: (a) the

---

[6]Measurement primacy definition: The majority of measurement data for evaluation and comparison will come directly from actual software coding and the run-time environment of PEPMA but not from a secondary data source.

three research questions presented previously, (b) six units of analyses, (c) a specific procedure of performance measurement system to obtain evidence, and (d), a method to verify the results.

For the Performance Measurement System (PMS) to be accurate and precise, a mathematical assessment in the projective domain is necessary. PMS's context will contain an interdependent group of leading metrics forming a unified whole, the KPI. Leading metrics will be obtained at three specific levels of evaluation: (a) exponentiation service, (b) point-doubling and point-adding functions, and (c) supporting mathematical software routines for point-doubling and point-adding functions.

To answer all three research questions and be able to verify the empirical results, the author proposes applying IEEE standards 982.1-1988 and 982.1-2005 to evaluate and construct the Key Performance Indicators, while tailoring the efficiency measurements based on academic research and industry practices.

In addition to standards IEEE 982.1 (1988) and 982.1-2005, ISO/IEC 15939 (2001) and ISBSG (2007) standards also provide direction for successfully implementing a measurement program. Although these standards do not directly provide a method of measurement, they provide guidance to identify, define, and improve processes to obtain metrics. By these international standards, the core measurement of PEPMA's performance can be facilitated by monolithic software measurement tools taking a set of measurements as input and producing metrics, formulas, and Key Performance Indicators (KPI) together with evaluation and analysis.

Unit of Analysis

There were six units of analyses involved in conducting research. They are listed in Table 4 below. Readers are referred to IEEE 982.1 (1988), Herrmann (2007), Laird et al., (2006), and Keyes (2005) for discussions of these first five units of analyses.

Table 4. Unit of Analysis

Unit of Analysis	**Formula**	**Reference/Comment**
Compliance Metric	CM	FIPS-140-2
Static Complexity	SCM	
Weighted Information Flow Complexity	WIFC	
Module Maturity Index	MMI	
Functional Metric	FM	
Efficiency Metric and Formulation	EMF	Has the most weighting toward KPI

The Efficiency Metric and Formulation (EMF) is the most important and difficult task in this research. The EMF has the highest weighting for KPI since the research topic focuses on computation efficiency. The major variables for each unit of analysis are summarized in Table 5 below.

Table 5. Unit of Analysis, EMF

Unit of Analysis	**Major Variable**	**Comment**
NSS PEPMA	APT, EF, PD, PA, PAT	
OpenSSL PEPMA	APT, EF, PD, PA, PAT	
Infinity Point		
Run-time Factor		Includes System Architecture, Compilation Environment, Test Vectors
Performance Hardware Counter	Instruction Counter	Will call PAPI Services
Program Profiling	Instruction Soft Counter	Will load BOCHS Emulator
Formulation Analysis		Manual analysis steps

Note: APT, EF, PD, PA, PAT respectively stands for Affine to Projective Transformation, Exponentiation Function, Point Doubling, Point Adding, and Projective to Affine Transformation

The empirical evaluation method was used to provide a framework for research in the area of efficiency metrics and formulations. The first two units of analyses shown in Table 5 embrace two specific implementations used for the empirical study: NSS PEPMA and OpenSSL PEPMA. These two open-source implementations provided necessary procedures to execute: (a) projective transformation, (b) the exponentiation function, (c) point-doubling and point-adding computations in Jacobian's transformed domain, (d) modulo arithmetic (including 521-bit NIST modulo reduction), and (e) localized mathematical procedures. Thus, a large portion of the analysis was focused on these five sub-units (a-e) with specially chosen test vectors. Readers are referred to Appendix F for a listing of test vectors.

In addition, the definition of an infinity point will formalize how point-doubling and point-adding function can handle infinity coordinates in projective geometry.

The unit of analysis belonging to a runtime environment will identify the system architecture. Analysis of this particular unit was directed toward internal characteristics of the target CPU and its arithmetic unit. The results of the CPU characteristics partially contributed to runtime factors in the performance equation.

NSS PEPMA and OpenSSL PEPMA were written in C language; hence, the compilation environment and C compiler option settings will introduce some variations in the resulting code. The performance equations should record these characteristics as one of their performance coefficients so that the final result can be more defined.

OS overhead, threading time, and delay due to processor interrupt services are run-time factors. They might affect the cost index produced from the Performance Hardware Counter or Program Profiling process. These run-time factors are categorized as Quality

of Service (QoS) for the verification procedure under the targeted Operating System. For that reason, the evaluation used a low-overhead, 64-bit x86 Community Enterprise Linux Operating System, version 6.4 (CentOS) for performance analysis. This specific OS is stable; and the low-overhead helps decrease the error induced by the Performance Hardware Counter. However, since these QoS run-time factors are a complex subject, they were excluded from the report.

The Performance Hardware Counter and Program Profiling through Emulation will assist and provide verification for the results during the development of efficiency formulations. Mainly, the outcomes from Performance Hardware Counter and program profiling data will contribute to part of the verification process. For a more detailed description of Performance Hardware Counter or Program Profiling through Emulation, readers are referred to open-source PAPI (2013) or BOCHS (2013).

Compliance Metric

The Compliance Metric (CM) of PEPMA measures the compliance with FIPS-140-2 (FIPS-140-2, 2001; Herrmann, 2007, p. 91).

Static Complexity Metric

The Static Complexity Metric (SCM) measures the complexity of NSS or OpenSSL PEPMA's software modules (IEEE 982.1, 1988, p. 23; IEEE 982.2, 1988, p. 60).

$$SCM = E - N + 1$$
$$SCM \approx RG$$

where
E = number of edges
N = number of nodes
RG = number of software modules bounded by edges with no edges crossing

Weighted Information Flow Complexity

The Weighted Information Flow Complexity (WIFC) measures inter-module complexity. The local direct flow exists if either PEPMA module invokes a second module and passes information to it, or the invoked PEPMA module returns a result to the caller (Herrmann, 2007, p. 121; IEEE 982.2, 1988, p. 74).

$$\text{WIFC} = (\, fanin \times fanout)^2 \times length$$

where:
fanin = Local flows into module + number of data structures from which the module receives data
fanin = Local flows out of module + number of data structures that the module outputs
length = Number of source statement in the module

Module Maturity Index

The Module Maturity Index (MMI) measures the effect of changes from one software module baseline to the next. The effect of these changes will solely be directed toward the efficiency of PEPMA. The MMI will be derived with different compiler optimizing option settings based upon a general discussion in (Herrmann, 2007, p. 121), as originated in particular standards (IEEE 982.1, 1988, p. 19; IEEE 982.2, 1988, p. 51), or as described in other standards (IEEE 982.1, 2005, p. 26).

$$\text{MMI} = \frac{(M_T - F_C)}{M_T}$$

M_T = Number of modules in current baseline
F_C = Current baseline that includes changes from previous baseline

Functionality Metric

The Functionality Metric (FM) measures the consistency and interoperability between available point-doubling and point-adding functions. As noted, there are several different approaches currently available to construct point-doubling and point-adding functions in a projective domain. General discussions of this metric are found in (IEEE 982.2, 1988, pp. 70-71).

Efficiency Metric and Formulation

Efficiency measurement and formulation will be one of most significant aspects beside other Key Performance Indicators as presented previously. The efficiency will be measured with respect to its main objective, which is a minimization of computing costs to reduce the number of CPU instructions. To date, there have not been significant standards available for evaluating the efficiency of PEPMA; general discussions on a comparable topic are suggested in some IEEE sources (IEEE 982.1, 1988, pp. 33-34; IEEE 982.2, 1988, pp. 33-34, 91-93). Related information to address techniques used in Efficiency Metric and Formulation is also scattered in Keyes's literature (Keyes et al., 2005) and many other research papers presented in the literature review sections.

Additionally, because there is not an absolute reference that PEPMA's efficiency measurement can be based on, the reference for measuring PEPMA's efficiency will be relative ─ meaning in between efficiencies of NSS and OpenSSL. Essentially, the Efficiency Metric and Formulation (EMF) will be derived from analysis of computing procedures and counting the execution of units of measurement while applying specific test vectors. The sections below further define the sub-units of analysis for obtaining EMF.

NSS PEPMA

Exponentiation Function:

In a 2013 open-source release, Network Security Services (NSS, 2013) applied a 4-bit window on the scalar *k* in PEPMA's exponentiation service. This service is shown as a computation loop ③ in Figure 18 below.

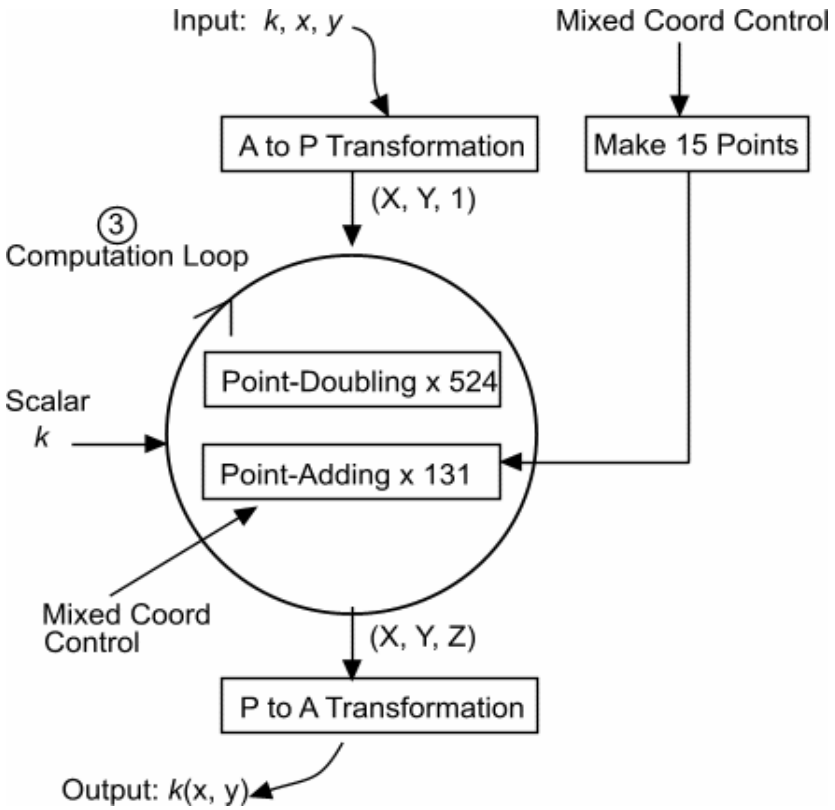

Figure 18. 4-bit Windowing Exponentiation Service

The NSS PEPMA computation makes 524 calls to the point-doubling and 131 calls to the point-adding function (NSS-2, 2014). Readers are referred to Appendix G for examining an exact number of calls. The 4-bit exponentiation windowing requires a pre-computing

of 15 Elliptic-curve points (pre = before entering exponentiation loop ③). The 15-point pre-computation calls point-doubling or point-adding services to calculate $k(x, y)$ using k = 2 to 15, and the coordinates (x, y) are the base coordinates of the cyclic subgroup of the chosen Elliptic curve. When k = 1, the pre-comp coordinates are actually the base point itself; thus, it requires no computation, just storing the coordinates in the table.

During the exponentiation computation in loop ③, k slides from right to left (bottom to top as shown) and 4 bits are extracted for indexing into the PRE-COMP table. The PRE-COMP value $p(x, y)$ will be used for point-adding if the index is non-zero (1...15); otherwise, a zero-value table index will signify a "No-Add" condition. The 15-point, pre-computing function makes service calls to 1 point doubling and 13 point-adding functions to completely fill the 15-point recomputed table.

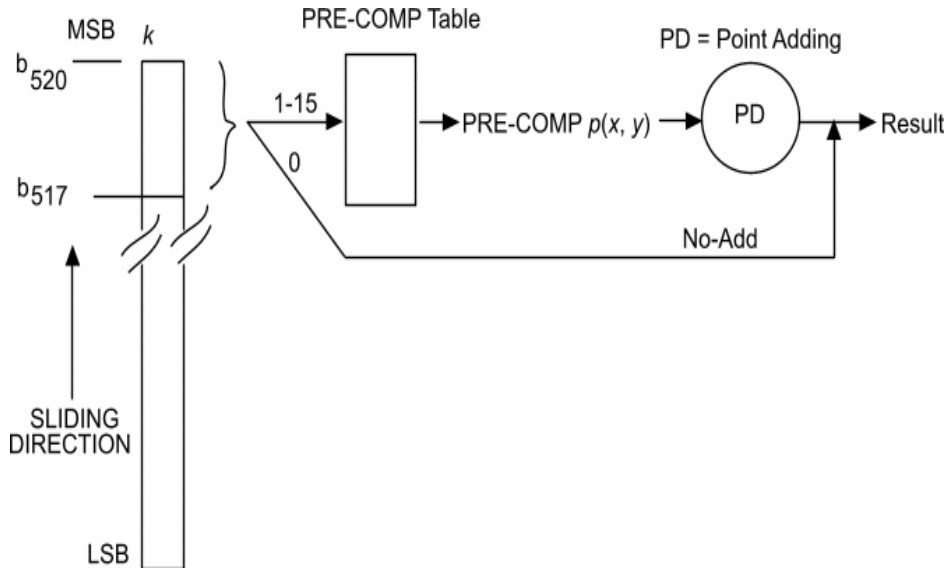

Figure 19. 4-bit Pre-comp Indexing Method

The mixed coordinate control signal directs the results from the 15-point pre-computing function to output the coordinates of type hybrid (mixed coordinates between affine and

projective coordinates). Building the pre-computed table is done outside the computation loop. The point-adding function then uses the 4-bit window taken from *k* to index into the table without the need to call point-adding four times. This reduces calling the point-adding function by 4:1 (131 × 4 = 524).

Point-doubling:

Inside the 4-bit windowing exponentiation service, NSS implemented a software function point-doubling $R(X_3, Y_3, Z_3) = 2 \times P(X, Y, Z)$ using weighted projective transformation (WPT) as described by (Cohen et al., 1998). In this document, the affine coordinate variable *x* is substituted with X/Z^2, and the affine coordinate variable *y* is substituted with Y/Z^3. These substitutions yield formulas for the point-doubling coordinates (X_3, Y_3, Z_3) as follows:

$$\text{Let } S = 4XY^2,\ M = 3X^2 - 3Z^4,\ T = M^2 - 2S \quad (1)$$
$$X_3 = T \quad (2)$$
$$Y_3 = -8Y^4 + M(S - T) \quad (3)$$
$$Z_3 = 2YZ \quad (4)$$

Based on a source-code written in C language and publicly released in 2013, any computing platform executing NSS point-doubling codes requires 4M+4S+5A+4Su+1Sh operations, where the arithmetic operators are designated as M=multiplying, S=squaring, A=addition, Su=subtraction, and Sh=Shift. Although the modular reduction routine calling is hidden from computing codes, it is actually called from inside at the end of each arithmetic operator (NSS, 2013). The computing cost index of Cohen yields 4M+6S, as compared to 4M+4S+5A+4Su+1Sh from NSS.

Point-adding:

Network Security Services (NSS) implemented a software function point-adding of point P_1 and P_2 using weighted projective transformation described by (Brown et al.,

2001). In their paper titled "Software Implementation of the NIST Elliptic Curves over Prime Fields," coordinate variable *x* is substituted with X/Z^2, and coordinate variable *y* is substituted with Y/Z^3, with the result R being $R(X_3, Y_3, Z_3) = P_1(X_1, Y_1, Z1) + P_2(X_2, Y_2, Z_2)$.

These substitutions yield the formulas as follows:

Let $A = X2Z1^2$, $B = Y2Z1^3$, $C = A - X1$, $D = B - Y1$ (1)
$X_3 = D^2 - (C^3 + 2X1\ C^2)$ (2)
$Y_3 = D (X1C^2 - X3)$ (3)
$Z_3 = Z1C$ (4)

Brown at al. (2001) recorded the arithmetic expenditure equal to 12M + 4S and excluded other arithmetic operations such as additions, subtractions and multiplications with constants. NSS actually executes a total of 8M+3S+2A+5Su.

Based on the explicit formulas above, NSS developers certified coding under FIPS 140-2 level 1 and released the point-adding function with C codes.

OpenSSL PEPMA

Exponentiation Function:

In a 2013 open-source release, OpenSSL applied a 5-bit window on the scalar *k* in PEPMA's exponentiation service shown as a computation loop ③ in Figure 20 below.

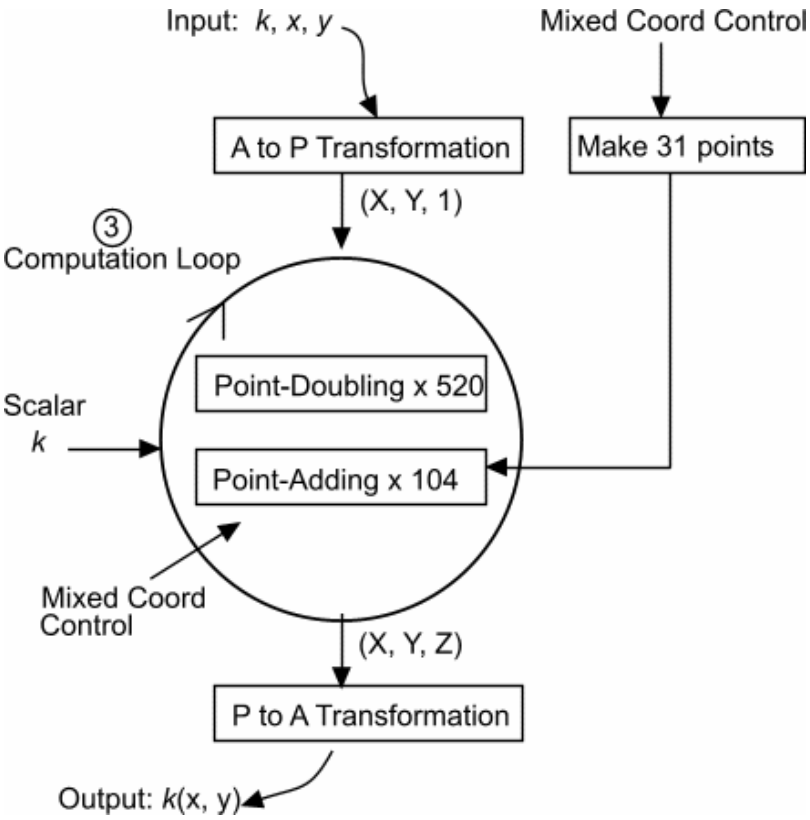

Figure 20. 5-bit Windowing Exponentiation Service

OpenSSL PEPMA makes 520 calls to the point-doubling and 104 calls to the point-adding function (OpenSSL-2, 2014). Readers are referred to Appendix H for examining an exact number of calls. The 5-bit windowing requires a pre-computing of 31 points using $k = 1$ to 31 and the base coordinates taken from the cyclic subgroup of the chosen

Elliptic curve. The 31-point pre-computing function makes service calls to 1 point doubling and 29 point-adding functions to fill the 31-point, pre-computed table. The point-adding function then uses the 5-bit window taken from *k* to index the table without the need to execute point-adding five times. This reduces calling the point-adding function by 5:1 (104 × 5 = 520).

Point-doubling:

Inside the 5-bit windowing exponentiation service, OpenSSL PEPMA implemented a software function point-doubling $R(X_3, Y_3, Z_3) = 2 \times P(X, Y, Z)$ using weighted projective transformation as described by (Brown et al., 2001). In this document, the affine coordinate variable *x* is substituted with X/Z^2, and the affine coordinate variable *y* is substituted with Y/Z^3. These substitutions yield the formulas for the point-doubling coordinates (X_3, Y_3, Z_3) as follows:

$$\text{Let } A = 4X_1Y_1^2,\ B = 8Y_1^4,\ C = 3(X_1 - Z_1^2)(X_1 + Z_1^2),\ D = C^2 - 2A \quad (1)$$

$$X_3 = D \quad (2)$$

$$Y_3 = C(A - D) - B \quad (3)$$

$$Z_3 = 2Y_1Z_1 \quad (4)$$

Based on a source-code written in C language and publicly released in 2013, any computing platform executing OpenSSL codes will consume 3M+5S+3A+4Su operations, where the arithmetic operators are designated as M=multiplying, S=squaring, A=addition, Su=subtraction. Although calling to the modular reduction routine is hidden from computing codes, it is actually called from inside at the end of each arithmetic operator. The computing cost index of Brown et al yields 8M+3S, as compared to 3M+5S+3A+4Su from OpenSSL.

Point-adding:

OpenSSL implemented a software function point-adding of point P_1 and P_2 using weighted projective transformation as described by one paper (Brown et al., 2001). In the paper, coordinate variable *x* is substituted with X/Z^2, and coordinate variable *y* is substituted with Y/Z^3, with the result R being $R(X_3, Y_3, Z_3) = P_1(X_1, Y_1, Z1) + P_2(X_2, Y_2, Z_2)$.

These substitutions yield the formulas as follows:

Let $A = X2Z1^2$, $B = Y2Z1^3$, $C = A - X1$, $D = B - Y1$ (1)
$X_3 = D^2 - (C^3 + 2X1\ C^2)$ (2)
$Y_3 = D (X1C^2 - X3)$ (3)
$Z_3 = Z1C$ (4)

Brown at al. (2001) recorded the arithmetic expenditure equal to 12M + 4S and excluded other arithmetic operations such as addition, subtraction, and multiplication with constants. NSS actually executes a total of 8M+3S+2A+5Su.

Based on the formulas above, OpenSSL developers certified coding under FIPS 140-2 level 1 and released the point-adding function with C-language source codes.

Point at Infinity

Based on point computation in the projective domain, there will be no use of the projective coordinates at (X = 0, Y = 0, Z = 0). These particular projective coordinates will be used as variable labels for a specific point at infinity. During mathematical processing, this zero-vector will be detected and subsequently called for a software handler to take care of the point at infinity.

Performance Hardware Counter

The purpose of using the Performance Hardware Counter is to approximate PEPMA machine instruction counts. This metric is available by calling the Performance Application Programming Interface, PAPI. Both PEMA and PAPI will run under host OS in real-time; thus, there will be synchronization issues and activation of filtering to address multiple accesses into the Performance Hardware Counter.

Program Profiling and Emulation

The purpose of using Program Profiling and Emulation is to obtain precise and accurate PEPMA counts of executing machine-codes. This metric is made available by activation of the BOCHS hardware emulator (BOCHS, 2013). In turn, BOCHS will supply a virtual-machine runtime environment to PEPMA. In this virtual-machine setup, the Operating System CentOS 6.4 is the host OS that provides a virtual environment to the guest OS, which is also a Centos 6.x OS. PEPMA runs under the Guest OS. The diagram in Figure 21 shows a structure of Program Profiling and Emulation in relation to other OSes, a Synchronization Agent, Software Counters, and PEPMA itself.

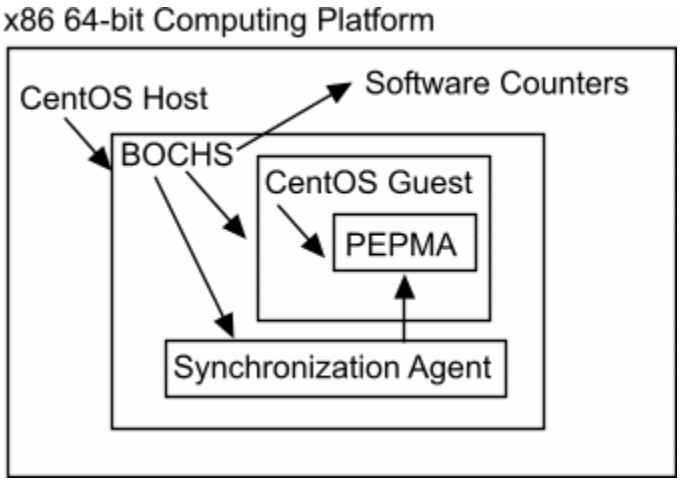

Figure 21. BOCHS Hardware Emulation

The Synchronization Agent filters the commands to BOCHS to adjust for access into each machine-code emulation. When the code in PEPMA's executable file accesses the emulated machine-code, the precise and accurate number of accesses will be recorded in Software Counters. An example of the emulation workflow is shown below:

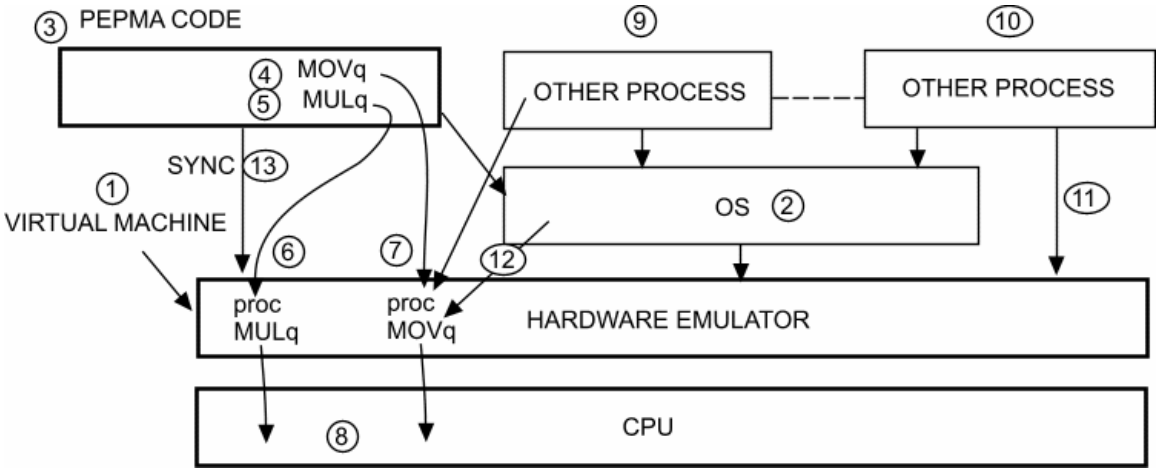

Figure 22. Accurate Efficiency Evaluation of PEPMA

The virtual machine ① loads the entire Guest Operating System ② without any modification to the Guest OS executable binary image. The Guest OS then loads and executes PEPMA code ③ without modification to the PEPMA executable binary image. Inside PEPMA code ③, the executable CPU instructions – for example, MULq ④ or MOVq ⑤ – will call the procedure "*proc* MULq" ⑥ or "*proc* MOVq" ⑦ at the hardware emulator. These two procedures will emulate the CPU instruction MULq ④ or MOVq ⑤. In turn, procedures ⑥ and ⑦ will call the hardware CPU ⑧ to fulfill the machine-code emulation. However, the other processes ⑨ and ⑩ can also call "*proc* MOVq" from the hardware emulator to emulate MULq ④ or MOVq ⑤. Because the emulated CPU instructions can be called from multiple processes, emulation of PEPMA codes must synchronize with the hardware emulator via the communication path ⑬ to obtain consistent and correct counting of the execution of machine-codes from PEPMA.

Run-time Factors

System Architecture:

Associated environments of PEPMA are factors that could potentially change the efficiency indexes of PEPMA in run-time. Thus, without accounting for these factors, the comparison between two PEPMAs might not be accurate. One influential environmental factor is the use of digit representation. As presented in the literature review chapter, a digit contained in a complete big number is usually referred to as a limb – a computation unit composed of several bits that should fit into a chosen system architecture. Otherwise, computational efficiency might suffer.

Operating System overhead, pipe-line queuing, memory cache, threading lost time due to other processes, and system interrupts overhead are factors that affect the run-time environments of PEPMA. These run-time environments exist but will not be considered in this research. Background of these factors can be found in computer architecture and quantitative experimental analyses from these references (Hennessy, 2006; Szerwinski, 2008).

Compiling Environment:

Other influential environmental factors are compiling options. Today, NSS, OpenSSL, and other open sources are mostly written in high-level languages. Given the different compiler option settings, the compilation of high-level languages will end up with different run-time machine codes.

Domain Parameters:

Vector contents coming from the domain parameter are expected to contribute to the performance evaluation of PEPMA as well. Domain parameters from the Client or Server side are of the same tuple (p, a, b, G, n, h), where the product $n \times G(x, y)$ using the EPM method must equal to infinity point O of the Elliptic curve. The domain parameter n is the order of the subgroup, and h is the cofactor equal to the size of the cyclic subgroup divided by n. The descriptions of these parameters are found in (FIPS PUB 186-4, 2013, p. 16).

Test Vectors:

A global third-party laboratory, which is accredited as Cryptographic and Security Testing (CST), can provide validation testing for FIPS approved and NIST recommended cryptographic algorithms and components of algorithms. A description of the validation program for cryptographic algorithms (CAVP) can be found at the NIST website (CAVP, 2013). Within the body of CAVP, NIST has approved the use of test vectors for Elliptic Curve Digital Signature Algorithm (ECDSA) as specified in ANSI X9.62 (ANSI, 2005) to informally verify implementation. To keep efficiency measurement consistent across the verification platform, NIST-recommended test vectors for ECDSA which will be applied toward the comparison between NSS and OpenSSL PEPMA.

Another recommendation for the test vector is posted in NIST Special Publication 800-56A (NIST 800-56A, 2013). The older version test vectors for ECDH are also available from Certicom Research (Certicom, 1999), also known as Standards for Efficient Cryptography organization (SEC). These test vectors will also be applied in the comparison between NSS and OpenSSL PEPMA to explore the inconsistency between

measurements of efficiency due to the application of different test vectors. Readers are referred to Appendix F for NIST recommended test vectors.

Rigorous evidence of characteristics of the test vector may be found in Elliptic-curve and projective transformation literature (NIST, 2010; Certicom Research, 2009; Blake, 2001; Menezes et. al., 1996; Cohn, 1962) and Finite-Field Projective Geometry (Rosen, 2006). Following suggestions from these papers, other possible test vector contents can be calculated from the order of the curve *n*, modulus *m*, and infinity point O. Readers are referred to Appendix D for the value of modulus *m* and the order *n* of cyclic subfield.

Efficiency Formulation Analysis

From a top-level formulation analysis, our adopted 12-step, closed-loop concept to generate formulations of new metrics and to verify formulas is shown in Figure 23 below. The overall technical approach included analysis, collected costs, and formulated the efficiency of these necessary procedures based on the actual number of machine-code instructions in a 64-bit x86 run-time environment. Subsequently, the formulation analysis did allow the development of quantifiable key performance indicators, which provided the benchmark in supporting realistic performance figures for PEPMA.

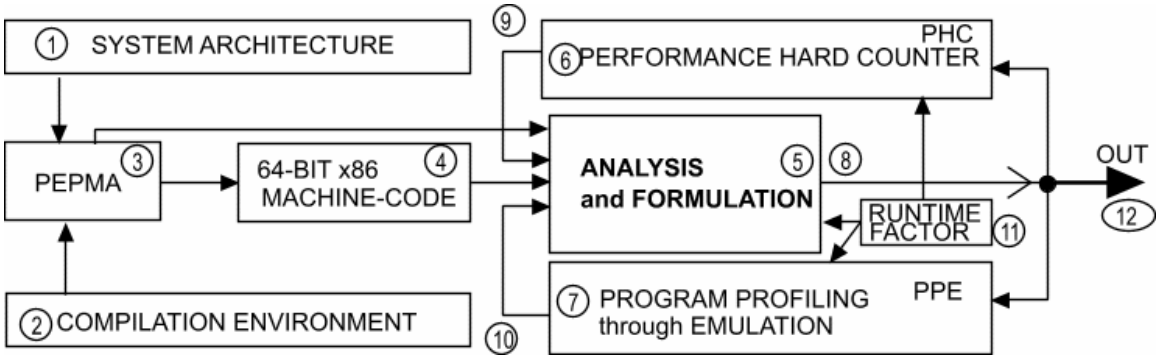

Figure 23. Formulation Analysis Block Diagram

The System Architecture ① and Compilation Environment ②, along with two open-sources – NSS and OpenSSL PEPMAs ③ – will generate a 64-bit x86 Machine Code ④.

Dependencies such as algorithms, looping, runtime factors, etc. from ③, ④ and ⑪ feeds Analysis and Formulation ⑤ to generate efficiency formulas. The analysis, design, development, and test of efficiency formulas occur in block ⑤.

Outcomes ⑧ from Analysis and Formulation will feed PHC, the Performance-Hardware Counter ⑥ and PPE, Program Profiling through Emulation ⑦ for comparison. Feedback paths ⑨ and ⑩ will adjust and verify formulations in the Analysis and Formulation block ⑤, which is the focus of this study. The work-flow approach for the analysis and formulation of block ⑤ will mostly be based on a deductive-reasoning model. An example of the work-flow for formulations of NSS PEPMA is shown in Figure 24 below:

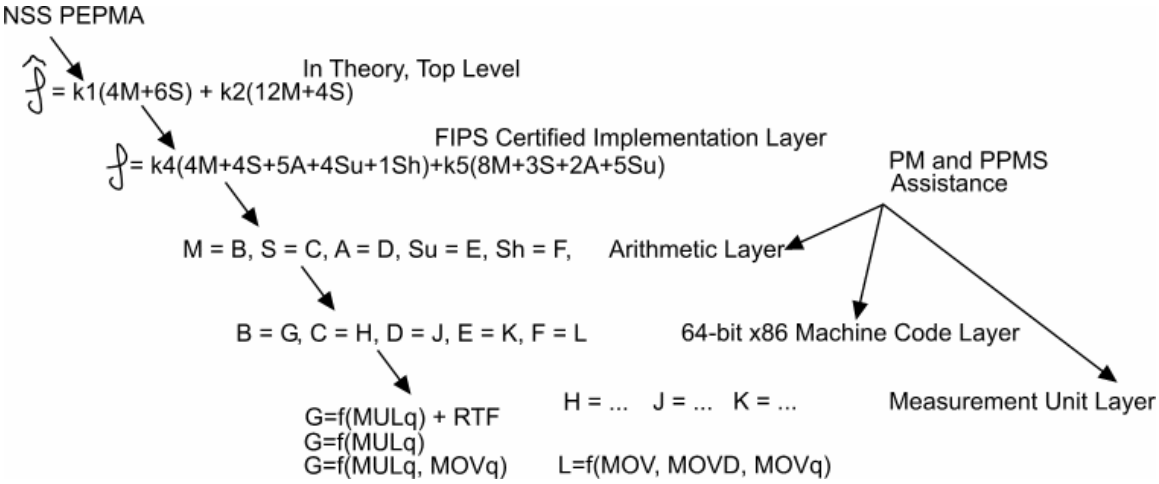

Figure 24. An Example of Performance Formulation

From the top-level, we derive two PEPMA formulas: one in theory (Cohen et al., 2006; Brown et al., 2001) and one with actual implementation (NSS, 2013). If the arithmetic operators are designated as M=multiplying, S=squaring, A=addition, Su=subtraction,

Sh=Shift bit, then the performance formulation of NSS PEPMA might have five new metrics: A, Su, Sh, k_4, k_5, etc. Therefore, the formula might be:

$$f = \mathrm{k}_4(4\mathrm{M} + 4\mathrm{S} + 5\mathrm{A} + 4\mathrm{Su} + 1\mathrm{Sh}) + \mathrm{k}_5(8\mathrm{M} + 3\mathrm{S} + 2\mathrm{A} + 5\mathrm{Su})$$

From the formula above, the equations for M, S, A, Su, Sh, k_4, k_5 in terms of algorithms, methods, looping, modulo reduction, test vectors, runtime factors, etc. can be derived. The equation M and its coefficients might have a form:

$$\mathrm{M} = f\ (\mathrm{method}, \mathrm{looping}, \mathrm{modulus}, \mathrm{testVectors}, \mathrm{RTF}) = \mathrm{B}$$

From the metrology requirement B, formulas G in terms of how many 64-bit x86 machine codes are required to accomplish function M can be derived. At the bottom-level, performance functions G, H, J, K and L will have the formulations in terms of machine-code instructions, such as MULq or MOVq, as units of measurement.

Complications will arise at the Arithmetic Layer, the 64-bit-x86-Machine-Code Layer, and the Measurement Unit Layer. The Performance Hardware Counter ⑥ and Program Profiling through Emulation ⑦ instruments will help fine-tune and verify the formulations at these layers.

Point ⑫ in Figure 23 indicates an exit path for this technical approach, where the outcomes will delineate final descriptions, comprehensive analysis, numeric presentations, and computing cost formulations for this study.

Individual performance comparisons of computing procedures (*f* hat, *f*, A, B, C, D...L etc.) will help software developers choose better projective computation and superior underlying mathematical service routines for the implementation of PEPMA. Subsequently, combining these quantifiable metrics into a single key performance indicator will offer ways to finally improve projective scalar point-multiplication

technology. The results will offer users the ability to dynamically or statically select the most efficient PEPMA.

Method for Verification

Program Profiling through Emulation (PPE) and its internal Software Counters, along with Performance Hardware Counters (PHC), was used to verify the efficiency formulas. Readers are referred to (BOCHS, 2013; Code XL, 2013) for a description of PPE, and to (Intel PERC, 2013; PAPI, 2013; Levinthal, 2009; Drongowski, 2008) for a description of PHC. A flowchart of the verification method is shown in Figure 25.

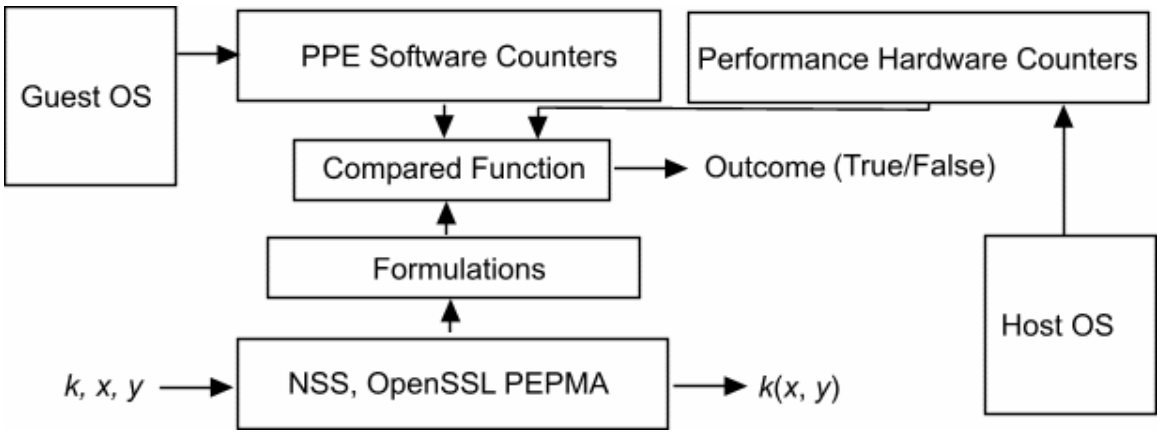

Figure 25. Efficiency Verification Block Diagram

Projected Outcome

The projected outcome will be the verification of the efficiency formulations with the prescribed method. Due to limited development resources, this research will not provide a verification method for the other five performance indicators: CM, SCM, WIFC, MMI, FM. Readers can reference industry practices for detailed descriptions of these five KPIs. Industry practice recommendations can be found at these cited sources (Herrmann, 2007; Hennessy, 2006).

Proposition of Format for Presenting the Results

The Performance Measurement System (PMS) used in this research will produce an interdependent group of leading performance indexes forming intermediate and final performance indicators. Table 6 shows an example of the final result.

Table 6. KPI between NSS and OpenSSL PEPMA

KPI	**Max Value**	**OpenSSL Score (Target Value)**	**NSS Score (Unit Under Test)**	**Weight %**	**Subtotal**
CM				2	2
SCM				2	2
WIFC				3	5
MMI				5	10
FM				3	5
EMF	100	85	90	80	72
Total		90		100	95

The proposed PMS will obtain six Key Performance Indicators (KPI) and post them in a table along with their weighting factor. The EMF's score for NSS PEPMA, the Unit Under Test, for example, will be 90. Since the weight of this EMF KPI is 80% of the KPIs, then NSS PEPMA scores 72, as shown in the subtotal column. Presumably, after summing all KPIs in the column subtotal, NSS is given a score of 95, which is higher than the OpenSSL target value of 90. This implies that NSS PEPMA is a better Projective Elliptic-curve Point Multiplication Agent. Revisiting the EMF formulas will offer insights to improve PEPMA's computing efficiency.

As shown in Table 6, formulations for the top five KPIs – CM, SCM, WIFC, MMI, FM – are described in the previous sections. The formulas for efficiency will likely have the following structures:

$$EMF_1 = k_a(k_1 MULq + k_2 MOVq) + k_b(k_3 MULq + k_4 MOVq) - RTF_1 + ...$$

$$EMF_2 = k_c(k_5 MULq + k_6 MOVq) + k_d(k_7 MULq + k_8 MOVq) - RTF_2 + ...$$

$$EMF_K = ...$$

$$EMF_{NSS} = \sum_{N=1}^{K} EMF_N$$

Labels MULq or MOVq are the anticipated units of measurement. Each element of the equation is the efficiency (or cost) of the mathematical module servicing PEPMA. Each EMF value will be normalized.

Values in Table 6 can be applied toward a Combined Key Performance Indicator. Its formula is defined as follows.

Combined Key Performance Indicator

The Combined Key Performance Indicator (CKPI) is defined as follows (Herrmann, 2007, pp. 123-124):

$$CKPI = \sum_{i=1}^{6} KPIi$$

KPI_1 = 0 if accuracy goals are not met
KPI_1 = 1 if accuracy goals are met
KPI_1 = 2 if accuracy goals are exceeded

KPI_2 = 0 if precision goals are not met
KPI_2 = 1 if precision goals are met
KPI_2 = 2 if precision goals are exceeded

KPI_3 = 0 if response-time goals are not met
KPI_3 = 1 if response-time goals are met
KPI_3 = 2 if response-time goals are exceeded

KPI_4 = 0 if memory-utilization goals are not met
KPI_4 = 1 if memory-utilization goals are met
KPI_4 = 2 if memory-utilization goals are exceeded

KPI_5 = 0 if storage goals are not met
KPI_5 = 1 if storage goals are met
KPI_5 = 2 if storage goals are exceeded

KPI_6 = 0 if storage goals are not met
KPI_6 = 1 if storage goals are met
KPI_6 = 2 if storage goals are exceeded

KPI_7 = 0 if transaction processing rates are not met
KPI_7 = 1 if transaction processing rates are met
KPI_7 = 2 if transaction processing rates are exceeded

Resource Requirements

NSS and OpenSSL implementations of PEPMA will run under a 64-bit Linux based Operating System. Particularly, the 64-bit OS will be the Community ENTerprise Operating System version 6.4 (CENTOS). The GCC version 4.4.7.3 compiler from Redhat Linux will be used for the compilation of NSS and OpenSSL PEPMA C codes to x86 assembly language, and then onto 64-bit x86 machine-code instructions.

Timeline

The following was the proposed time line toward the completion of this study:

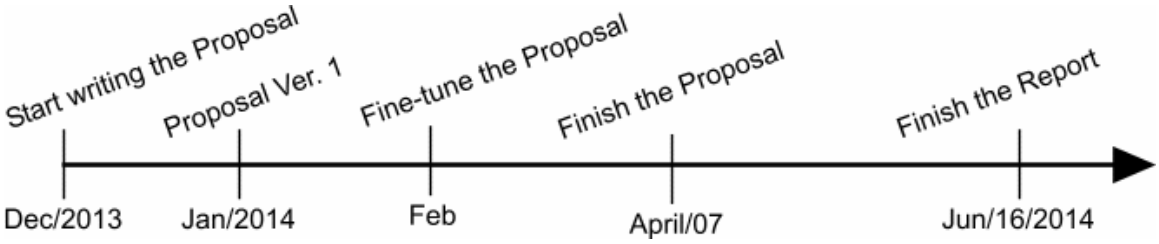

Figure 26. Timeline

Chapter Summary

The construction of a precision Performance Measurement System for PEPMA will be required to accurately evaluate the efficiency of PEPMA and provide a formal approach to improve the performance of the system measured. The beginning of the approach is a proposal for obtaining leading performance indexes that can be constructed at three specific levels of evaluation: (a) exponentiation service, (b) point-doubling and point-adding functions, and (c) supporting mathematical software routines for point-doubling and point-adding functions. These three assessment levels, six units of analyses, a specific comparative method with BOCHS and PAPI for the verification of results, and the manual formulation analysis will all help reach the final goal while IEEE standards will help to construct Key Performance Indicators.

Chapter 4

Results

Introduction

This chapter reports the findings, the associated formulas (if any), and presents data analysis of such findings. The findings consisted of outcomes, which were discovered during close examination of six units of analyses directed toward the performance comparison between NSS and OpenSSL PEPMA in 64-bit x86 runtime environment. The units of analyses are listed in Table 7 below.

Table 7. Finding of Six Units of Analyses

Unit of Analysis	**Formula**
Efficiency Metric and Formulation	EMF
Compliance Metric	CM
Weighted Information Flow Complexity	WIFC
Cyclomatic Complexity Metric	CCM
Functional Metric	FM
Module Maturity Index	MMI

The data analysis section in this chapter provides information to familiarize the reader with the basis of the finding. The chapter concludes with a summary of all findings and data analyses, preparing the readers for the final chapter.

The ultimate goal of this research with two FIPS-140-2 certified studying cases was to develop a repeatable and deterministic evaluation approach of the performance of PEPMA. The study provides a detailed framework for the evaluators to construct a better evaluation method. Thus, the final contribution to the field of cryptography is the formal evaluation method that can lead to the performance improvement of PEPMA. Other benefits related to industrial applications in the field will not be discussed in this chapter.

They will be discussed in the closing conclusion, implication, recommendation and summary of the final chapter.

Systematic Software Reviews and Selection of Unit of Analysis

The software reviews in this study adhered to the code walk-through and software inspection formal process as recommended in IEEE 1028 (2008). The purpose of the review is to determine and put together the performance improvements through the findings and data analysis. IEEE-1028 covered code walk-through along with software inspections first appeared in 1988 and then 1997 (IEEE 1028, 1997). This standard suggested six reviewing areas of software products and provided ways to identify anomalies, including errors and deviations from standards and specifications. However, it is important to note that this research does not intend to identify and correct the implementation errors; multiple comprehensive reviews across six areas have been accomplished at the product design phase and/or at an accredited FIPS certification site. Leveraging the same walk-through and code inspection procedures recommended in IEEE standards, this study aims to provide ways for code improvements through findings and data analysis as shown in Figure 27.

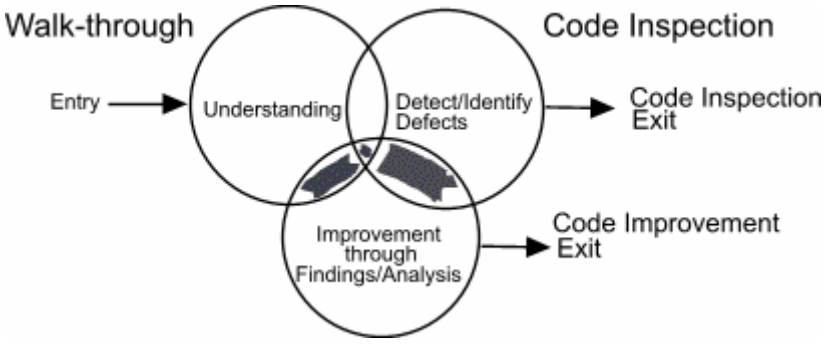

Figure 27. Types of Software Review Used in the Research

The shaded areas in Figure 27 show an entry and exit path to obtain potential code improvement portions which the formal code walk-though or inspection procedure possibly might have missed or was not intended to correct.

While Fagan developed a formal software inspection process at IBM in the mid 1970s focusing on finding software defects, his work also resulted in a schematic of defect classification and distribution (Fagan, 1976). However, the detail of the classification was not clearly presented at that time. Later, Fagan's inspection methods were thoroughly discussed in software inspection by Gilb & Graham (1993), which focused on defect identification. According to Runeson et al (2006) or Jones (2010), defects can be classified in many different ways. First, a defect can be cataloged as either an omission (something is missing) or a commission (something is incorrect). Second, a defect may be defined based on technical contents as to whether the product meets or does not meet a specific requirement (i.e., efficiency or FIPS compliance). Third, defects may be categorized by the impact to the user as the result of technical capability running on a specific computing platform. During software review of PEPMA coding, this research seeks for the omission and presents it in the key performance indicators and improvements. The standard IEEE code walk-though and software inspection process were used to collect quantitative data at defined points on prior works in this area.

Although code walk-through and code inspection are two related software review methodologies, the latter is more formal than the first. Both walk-through and inspection focus on finding errors in the product but not correcting them. Code walk-through often requires less expertise in the subject domain while inspection might require professionally trained inspectors.

The software inspections remove roughly 85% of the total errors as Fagan reported in his studying cases. It has been shown no other techniques, walk-through, or testing using automatic checking tools identify errors better than the manual code inspection. Jones suggested that manual code inspection could potentially remove 70% of the total errors as he observed the outcomes in industrial projects (Jones, 2010, p. 574). It has been advocated in (Source-Selection, 2011) that the end users in the field might experience the remaining defects through the so called "degradation of performance." These user experiences pertain to the remaining 15% to 30% of anomalies not found by code walk-though or code inspections. The shaded area in Figure 27 shows an opportunity for improving efficiency that the code walk-though process or software inspection procedure at the product design phase, or at an accredited FIPS certification site, possibly missed or did not intend to correct.

The discussion above offers some hints that the code walk-through could help to promote the improvement of the product, as seen by a person with less expertise; meanwhile, code inspection could improve the product, according to the checklist of items to be examined. For example, if PEPMA code is to be manually inspected, the inspection checklist can include such items listed as six units of analyses pointing to the efficiency, standards compliance, or coding information flow complexity. For a general discussion of why these key performance indicators were selected, readers are referred to recommendations in the standard ISO/IEC 15939 (2001). To answer three research questions and be able to verify the empirical results, we suggested applying IEEE standards 982.1-1988 and 982.1-2005 to evaluate and construct six units of analyses while tailoring the efficiency measurements based on machine virtualization technology.

Limitation

Since the goal of this study was to develop a more rigorous understanding of a formal performance evaluation approach, the reader should bear in mind that all findings in this report are subject to several limitations. First, the findings presented here are only a representation of the essential outcome, which should provide meaningful evaluation results. As an example, Table 8 (below) lists eleven findings to provide adequate results for the evaluation of Cyclomatic Complexity Metric; however, one finding of the sub-module is needed to represent the idea adequately.

Table 8. Finding Limitation

Unit Under Test	**Sub-Module**	**Low-Level Routine**
Cyclomatic Complexity Metric	APT, EF, PD, PA, PAT	Adding, Subtraction, Modulo Reduction Squaring, Multiplication, Inversion

Second, findings of other units of analyses have been purposely excluded due to the limited scope of this research. Third, the limitations mentioned above are also applicable to the construct of formulations and data analysis. Fourth, only Efficiency Metric and Formulation contains verification methodology while the other five units do not. Fifth, the current research was not specifically designed to evaluate the importance factors of each Key Performance Indicator.

Because of these limitations, the comparison results should be interpreted cautiously; further investigation and report of the unlisted findings might be necessary to achieve a realistic goal. Furthermore, the emphasis of this research was to uncover whether the performance of PEPMA might be unknown based on existing theoretical work, and what metrics should be used to candidly evaluate PEPMA's efficiency. However, the findings and analyses of low-level mathematic routines are beyond the

scope of this results chapter. They are presented here for completeness; although, the findings and analyses of such low-level mathematic routines do offer more accuracy to the final product.

While the literature review and the methodology section provided some evidence to answer these research questions, the findings from six units of analyses could uncover concrete facts of whether the performance of PEPMA might be unknown based on existing theoretical work. Together with the findings, the constructed formulas and data analysis could further confirm which metrics can be used to truthfully evaluate PEPMA's efficiency.

For the benefits of applications in the cryptographic field, are there realistic and deterministic performance evaluation approaches which will enable the code implementers to improve PEPMA’s efficiency based on the empirical comparison? In this results chapter, the findings and results based on six units of analyses could also suggest a tangible setup for such a formal evaluation method.

Chapter Organization

This results chapter is organized into seven sections: ① to ⑦, corresponding to the six units of analyses, and a combined performance indicator as shown in Figure 28.

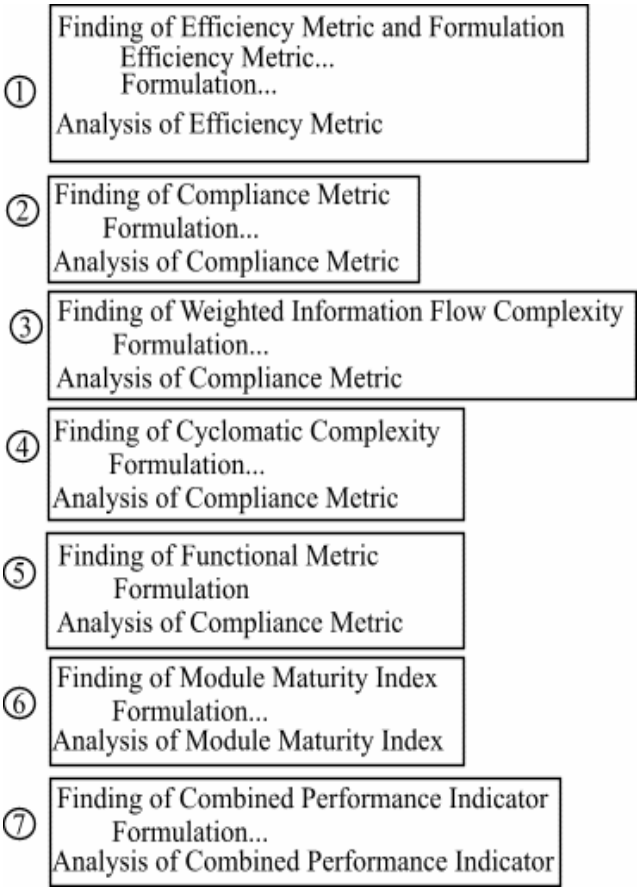

Figure 28. Formal Performance Evaluation Approach

Figure 28 also represents the sequential order of flow of a suggested formal approach for the performance evaluation where unit of analysis 1 – Finding of Compliance Metric and Formulation – carries the highest level of importance/weight; and unit of analysis 6 – Finding of Module Maturity Index – carries the lowest level of importance/weight. However, this research is not specifically designed to evaluate the importance factors of each Key Performance Indicator; hence the order of evaluation might change based on a case-by-case application where the level of importance/weight can change for each unit

of analysis. One possible approach to determine the importance level is found in (Source-Selection, 2011).

Verification of the Finding

As previously presented in the methodology chapter, the performance relating to computational efficiency shall be verified through a formal verification process using Program Profiling and Emulation with BOCHS. Additionally, the computational efficiency shall be verified through another formal verification process by acquiring the Performance Hardware Counter via Performance Application Programming Interface (PAPI). In this section titled the "Analysis of Efficiency Metric and Formulation," we applied these formal verification methods to fulfill the verification of the findings; hence, the verification will be reported thoroughly in the analysis section. This formal verification of efficiency will provide supports for a tangible closing conclusion of this research. The approach for verification is depicted in Figure 29 below.

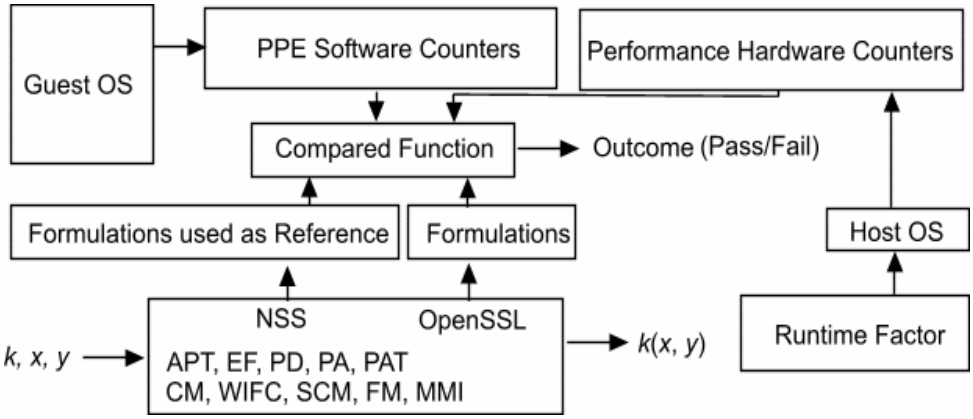

Figure 29. Efficiency Verification Block Diagram

Activation of virtual machine BOCHS to sandbox PEPMA under a guest Operating System was complex. It required a specific set of instructions and process synchronization in order to properly execute PEPMA under the virtual machine

environment. Readers are referred to Appendix K, the Operation of BOCHS, for more details on the commands and the setups of BOCHS.

Acquiring the performance hardware counter via Performance Application Programming Interface (PAPI) also required a specific setup and compilation. Readers are referred to Appendix L, the Operation of PAPI, for more details on the commands and the setups of PAPI.

Concept of Instrumentation

This section describes the concept and rationale using two measuring instruments PAPI and BOCHS. The instrument PAPI can accurately measure the total number of instructions, TOT_INS, which are required to process a particular unit-under-test (a unit-under-test may be any of sub-modules belonging to six units of analysis). However, the measurement TOT_INS is just a lump sum of all CPU instructions (around 26 millions for NSS PEPMA); this metric does not indicate what types of CPU instruction that the unit-under-test uses. Therefore, it is not a good metric for modular improvement (see Appendix U, V for the descriptions of metrics). On the other hand, the instrument BOCHS can accurately measure the total number of instructions; and it can also indicate what types of CPU instruction that the unit-under-test uses. If the cost for processing a module was approximately constructed by BOCHS as follows:

$$\text{MODULE\_COST} \approx k_1(\text{MULq}) + k_2(\text{MOVq})$$

then the exact formula of MODULE_COST must be

$$\text{MODULE\_COST} = k_1(\text{MULq}) + k_2(\text{MOVq}) + \text{OHF}$$

where the term OHF is defined as an overhead factor; and the OHF might include other MULq, MOVq, or other types CPU instruction.

If the metric TOT_INS is exact, then the following equality must be true

$$TOT\_INS = MODULE\_COST = k_1(MULq) + k_2(MOVq) + OHF$$

To be absolutely accurate, the equation TOT_INS must include all of the CPU instructions. For instance, two additional coefficients k_3(ADD) and k_4(SUB) in the equation TOT_INS will make the result more accurate:

$$TOT\_INS \approx k_1(MULq) + k_2(MOVq) + k_3(ADD) + k_4(SUB)$$

Due to limited scope of this study, the coefficient in TOT_INS equation does not expand beyond the first two CPU instructions. Thus, the expansion coefficients (k_3(ADD) + k_4(SUB) + *others CPU Instruction...*) are lumped sum into a single over-head factor, OHF. For comparison, one could convert TOT_INS to the total number of CPU cycles.

Overview of the Finding in General

Documentation search and/or certificates were used to collect some findings; however, the primary method for collecting the findings was through the examination of NSS/OpenSSL source codes. Additionally, the findings were discovered through running executable binaries under both host and guest Operating Systems (Virtual Machine using BOCHS) and taking the results from the program output messages. Since the results from six units of analyses directly contributed to the performance comparison, six Key Performance Indicators were derived from the following six units of analyses.

Table 9. Findings of Key Performance Indicators

Key Performance Indicator	**Formula**	**Importance Level (Note 1)**
Efficiency Metric and Formulation	EMF	6 = Highest
Compliance Metric	CM	5
Weighted Information Flow Complexity	WIFC	4
Cyclomatic Complexity Metric	CCM	3
Functional Metric	FM	2
Module Maturity Index	MMI	1 = Lowest

Note 1: The importance levels are only a representation/example. These levels were taken from a particular investigation of ECDH public-key exchange protocol used in the Department of Defense. This research was not specifically designed to evaluate the importance factors of each Key Performance Indicator. However, it has been advocated in the (Source-Selection, 2011) from the DoD suggesting approaches to obtain importance levels pertaining to technical risk of a product.

Overview of the Findings of Efficiency Metric and Formulation

The findings of the efficiency metric in this section is a collection of outcomes that have been discovered in examining computational efficiency directed toward the performance comparison between NSS and OpenSSL PEPMA in 64-bit x86 runtime environment. Essentially, this section shows the results of evaluating the components as illustrated in Figure 2, the Projective Elliptic-Curve Point-multiplication Agent in the introduction chapter. For reading convenience, Figure 2 has been expanded and shown here with three other units-under-test: ⑪ infinity point, ⑫ pre-computation table, and ⑬ runtime factor.

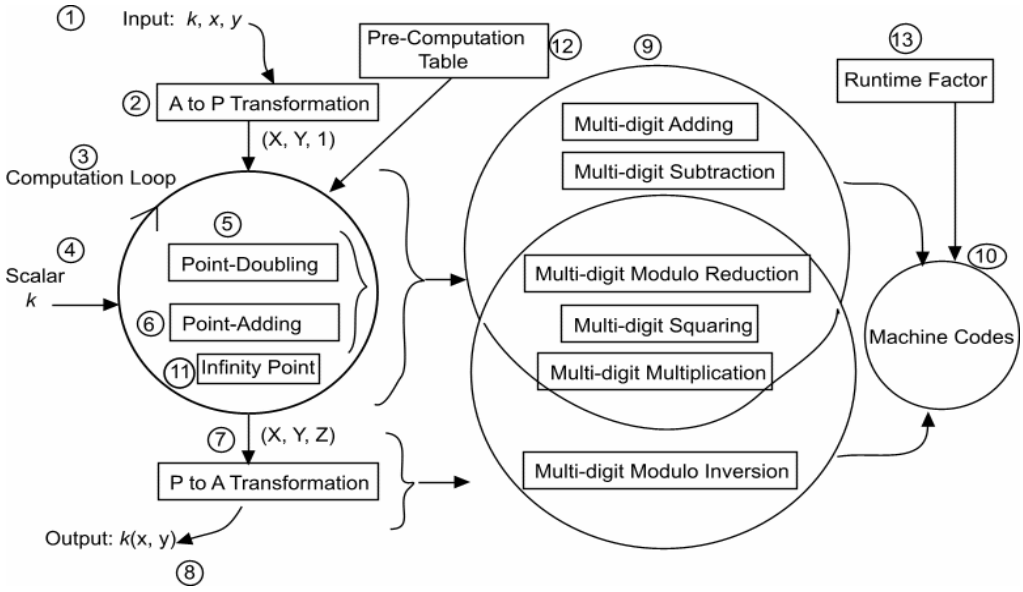

Figure 30. Projective Elliptic-Curve Point-multiplication Agent, Complete

The figure above illustrates the efficiency of NSS/OpenSSL PEPMA governed mainly by five sub-modules: Affine to Projective Transformation, Exponentiation Function, Point-Doubling, Point-Adding, and Projective to Affine Transformation (APT, EF, PD, PA, and PAT respectively). The efficiency of NSS/OpenSSL PEPMA also depends on how NSS/OpenSSL is implemented to handle the pre-computation, the infinity point, and the runtime factor.

Table 10. Finding of Efficiency Metric and Formulation

Unit Under Test	**Sub-Module**	**Low-Level Routine**
PEPMA	APT, EF, PD, PA, PAT	Adding, Subtraction, Modulo Reduction Squaring, Multiplication, Inversion
Pre-computation	PD, PA	Adding, Subtraction, Modulo Reduction Squaring, Multiplication, Inversion
Infinity Point		
Runtime Factor		

From Table 10, counting down from unit-under-test PEPMA, there were eleven findings: five counts for sub-modules (APT, etc.,) and six counts for low-level routines. Although the naming convention shown in the Sub-Module column and in the Low-Level Routine column is the same for both NSS and OpenSSL, sub-modules and low-level functional services comparing NSS PEPMA and OpenSSL PEPMA are not the same routines.

Furthermore, while all six low-level routines (listed in Table 10) fulfill the intended function, each sub-module might not need to call all six low-level routines; the explanation of which sub-level module calls which low-level routines is in order. Seven low-level routines for each sub-module are summarized in the last column of Table 10. The findings are presented in pairs (NSS APT vs. OpenSSL APT etc.) throughout the section for the convenience of reading when comparing NSS and OpenSSL cases. The

data analysis of this chapter includes only essential information to familiarize the reader with the basis of the findings. The formulations for computational cost are presented at the end of the findings section.

In the section “The Finding of Efficiency Metric and Formulation,” the word "Formulation" refers to the construct of more succinct computational cost formulations as the results of finding and data analysis. These formulations were used for verification of the findings, with the lowest units of measurement being MULq instruction and MOVq instruction, or scalar values. In the comparison of efficiency, NSS will serve as a reference point (compared OpenSSL against the results from NSS).

BOCHS and PAPI were the verification instruments that provide Program Profiling and Emulation Software Counters and Performance Hardware Counter, respectively. Based on such formal verification of efficiency, a more tangible closing conclusion can be drawn in the final chapter. The formal approach for verification of efficiency was previously illustrated in Figure 28.

The verification instruments worked with specially chosen test vectors. For a listing of test vectors used in this study, see Appendix F.

Infinity Point

In NSS, the representation of an infinity point in projective domain is defined as follows:

NSS infinity point $\triangleq$ (X, Y, 0) where X = don't care, Y = don't care

In OpenSSL, the representation of an infinity point in projective domain is defined as follows:

OpenSSL infinity point $\triangleq$ (X, Y, 0) where X = don't care, Y = don't care

Runtime Factor

OS overhead, threading time, and delay due to processor interrupt services are run-time factors. They might affect the cost index produced from the Performance Hardware Counter or Program Profiling process. These run-time factors are categorized as Quality of Service (QoS) for the verification procedure under the targeted Operating System. However, since these QoS run-time factors are a system specific subject, they were excluded from the findings and data analysis.

Finding of NSS Affine to Projective Transformation

The NSS Affine to Projective Transformation (APT) is a functional service routine that converts affine coordinates to the coordinate representations in the projective domain. File location and function calling conventions are listed in Table 11 below.

Table 11. Finding of NSS Affine to Projective Transformation

Finding of	**Description of APT**	**Comment**
File location	NSS\mozilla\security\nss\lib\freebl\ecl\ecp_jac.c	"NSS\" is the root directory where project was installed

Function	**ec_GFp_pt_aff2jac**
	mp_err ec_GFp_pt_aff2jac (const mp_int *px, const mp_int *py, mp_int *rx,
	mp_int *ry, mp_int *rz, const ECGroup *group)
	{
	... calling Sub-functions below...
	};
	
	Note:
	Description of data structure type "mp_int" can be found in the literature review section of low-level arithmetic representation
		*px, *py are the pointers to affine coordinates (*x*, *y*).
		
		*group points to a data structure having characteristics of the Elliptic-curve

Sub-function	mp_copy(px, rx);
	mp_copy(py, ry);
	mp_set_int(rz, 1);
		*rx, *ry, *rz are the pointers to the results in projective domain

As designed, the NSS APT function is conditionally called at the beginning of the Point-Adding function. The calling sequence is shown below:

```
Start computing PA:
        if (Z = = 0) ec_GFp_pt_aff2jac(...)
                        calling other functions...
```

Formulation

The formulations were derived by examining the operation of the following statements:

```
mp_copy(px, rx);
mp_copy(py, ry);
mp_set_int(rz, 1);
```

Table 12. NSS APT Formulation

Sub-Module	Formula	Unit of Measurement
NSS APT	NSS_APT ≈ 3(MULq)+29(MOVq)	MULq, MOVq
NSS APT	NSS_APT_PAPI_TOT_INS = 2745	All CPU Instructions
NSS APT	NSS_APT_PAPI_TOT_CYC = 7879	CPU cycles

For comparison, testing for efficiency of APT was repeated under two verification instruments, BOCHS and PAPI as shown in Table 13. The values shown in lower-limit (MIN), Typical (TYP), and upper-limit (MAX) are the accuracy ranges of the measuring instrument.

Table 13. NSS BOCHS/PAPI APT Limits

Components	MIN	TYP	MAX	Unit of Measurement
NSS APT BOCHS (Note 1)	3		3	MULq
NSS APT BOCHS	29		29	MOVq
NSS APT PAPI		2745		TOT_INS
NSS APT PAPI		7879		TOT_CYC

Note 1: Readers are referred to Appendix K, the Operation of BOCHS, and Appendix L, the Operation of PAPI, for more details on commands and setups of BOCHS/PAPI.

The same values presented in both "MIN" and "MAX" columns indicate that readings from instrumentation are exact. Readings presented in "TYP" column are not exact. They change from one sampling to the other.

Findings of APT in OpenSSL

The OpenSSL Affine to Projective Transformation (APT) is a service routine that converts affine coordinates to the coordinate representation in projective domain. File location and function calling conventions for OpenSSL are listed in Table 14 below.

Table 14. Finding of OpenSSL Affine to Projective Transformation

Finding of	**Description**	**Comment**
File location	O\crypto\ec\ec_lib.c	"O\" is the root directory where project was installed

Function	**EC_POINT_set_Jprojective_coordinates_GFp** (group, point, x, y, BN_value_one(), ctx);
	{
	.....
	group->meth->
	point_set_Jprojective_coordinates_GFp(group, point, x, y, z, ctx);
	.....
	}
	Note: group->meth->
	point_set_Jprojective_coordinates_GFp(...)
	is a name holder for function
	EC_POINT_set_Jprojective_coordinates_GFp
	"....." represents some other house-keeping functions
		*group points to an object having characteristics of the Elliptic-curve (as data), and points to executing functional pointers (as method). Thus, "group->meth->" is a pointer to an executing method.
		"point" is an array of *MDN*.
		BN_value_one() is a 521-bit *MDN* having a scalar value of 1.

Sub-function	**EC_POINT_set_Jprojective_coordinates_GFp** (const EC_GROUP *group, EC_POINT *point, const BIGNUM *x, const BIGNUM *y, const BIGNUM *z, BN_CTX *ctx)
	{
	.....
	calling group->meth->
	point_set_Jprojective_coordinates_GFp(group, point, x, y, z, ctx);
	.....
	};
		EC_GROUP,
		EC_POINT,
		BIGNUM,
		BN_CTX are data structures.
		"*ctx" is a pointer to a context database, a temporary and volatile holding data structure for the function

Table 15. Finding of OpenSSL Affine to Projective Transformation (Continued)

Finding of	Description	Comment
Sub-function	group->meth->point_set_Jprojective_coordinates_GFp(group, point, x, y, z, ctx); Note: group->meth->point_set_Jprojective_coordinates_GFp(...) is a name holder for the function below	

Sub-function	**ec_GFp_simple_set_Jprojective_coordinates_GFp**(const EC_GROUP *group, EC_POINT *point, const BIGNUM *x, const BIGNUM *y, const BIGNUM *z, BN_CTX *ctx)
	{
	.....
	BN_nnmod(&point->X, x, &group->field, ctx)
	BN_nnmod(&point->Y, y, &group->field, ctx)
	BN_nnmod(&point->Z, z, &group->field, ctx)
	.....
	}	

Sub-function	**BN_nnmod**(&point->X, x, &group->field, ctx) **BN_nnmod**(&point->Y, y, &group->field, ctx) **BN_nnmod**(&point->Z, z, &group->field, ctx)	**BN_nnmod**() reduces an *MDN* and places the result in "&point->Z"

Formulation

The formulation was carried out by executing the arithmetic operation of three statements with a specific test vector, and using BOCHS to read the results.

BN_nnmod(&point->X, x, &group->field, ctx)
BN_nnmod(&point->Y, y, &group->field, ctx)
BN_nnmod(&point->Z, z, &group->field, ctx)

Table 16. NSS APT Formulation

Sub-Module	Formula	Unit of Measurement
OpenSSL APT	OpenSSL APT ≈ 151(MULq) + 198(MOVq)	MULq, MOVq

For comparison, these measurements were repeated under two verification instruments, BOCHS and PAPI with test vector type A (see Appendix O).

Table 17. APT Comparison

Components	**NSS**	**OpenSSL**	**MAX**	**Unit of Measurement**
APT BOCHS	3	151		MULq
APT BOCHS	29	198		MOVq
APT PAPI	745	1626 *		TOT_INS
APT PAPI		4190 **		TOT_INS
APT PAPI	879	5052		TOT_CYC

* Test vector type A (see Appendix O). Computing platform type A (see Appendix R).

** Test vector type C (see Appendix Q) using CPU type C in a busy run-time environment (see Appendix T).

Analysis of Affine to Projective Transformation

The data gathered in Table 14, 15, 16 and 17 suggest that the computing time of target CPU for performing APT function is significantly different when comparing NSS and OpenSSL implementations. The data yielded by these findings provide convincing evidence that NSS implementation of APT might be more efficient since it uses three simple functions "copy" and "set" to set the values into the results of APT

```
mp_copy(px, rx);
mp_copy(py, ry);
mp_set_int(rz, 1);
```

while OpenSSL uses three modulo arithmetic routines BN_nnmod() to set three values into the results of APT. The cost of this computation depends on the content of the input test vector (x, y, z) and how efficient the modulo reduction arithmetic was done.

```
BN_nnmod(&point->X, x, &group->field, ctx)
BN_nnmod(&point->Y, y, &group->field, ctx)
BN_nnmod(&point->Z, z, &group->field, ctx)
```

For a detailed discussion of Affine to Projective Transformation, readers are referred to the Concept of Point Computation in Projective Domain, which was previously presented in the Literature Review chapter. Similarly, Ryabko et al. (2005) and Salomon (2006) have found that using a sub-function as shown in Table 11 would be more straightforward and better than using function BN_nnmod() as shown in Table 14, 15 for computation of Affine to Projective Transformation.

Finding of NSS Exponentiation Function

In an open-source version 3.12.4 release, Network Security Services (NSS, 2013) applied a 4-bit window on the scalar *k* in an exponentiation service. The Exponentiation Function (EF) is shown as computation loop ③ in Figure 31 below.

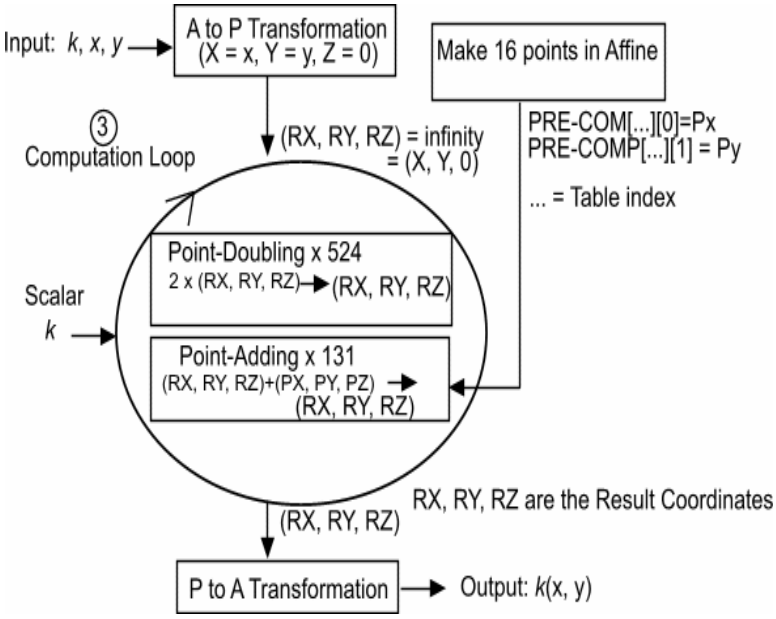

Figure 31. Exponentiation Function in NSS

The NSS PEPMA computation makes 524 calls to the point-doubling and 131 calls to the point-adding function (NSS-2, 2014). Readers are referred to Appendix G for examining the exact number of calls.

Real-time Samplings with PAPI, Function EF in NSS

The real-time sampling was taken from a particular desktop PC type B (see Appendix S) with test vectors type A (see Appendix O).

Table 18. Real-time Samplings, Function EF in NSS, Vectors Type A

Iteration	TOT_CYC	TOT_INS	Deviation of TOT_CYC from Minimum
1	15,094,162	26,707,442	49416
2	15,188,326	26,707,443	143,580
3	15,195,070	26,707,443	150,324
4	15,044,746	26,707,442	0
5	15,252,588	26,707,442	207,842

Figure below illustrates the deviations from iteration 4 of TOT_CYC as listed in Table 18.

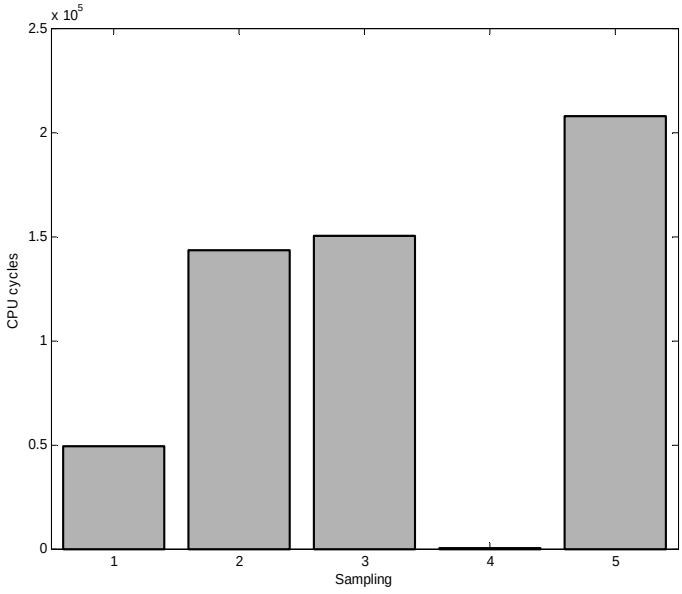

Figure 32. Real-time Samplings, EF in NSS, Test Vector Type A

Another real-time sampling was taken from the same desktop PC with test vectors type B (see Appendix P).

Table 19. Real-time Samplings, Function EF in NSS, Vectors Type B

Iteration	TOT_CYC	TOT_INS	Deviation of TOT_CYC from Minimum
1	14,842,668	26,248,085	179,771
2	14,669,926	26,248,085	7,029
3	14,662,897	26,248,085	0
4	14,743,208	26,248,085	80,311
5	14,693,759	26,248,087	30,862

Figure below illustrates the deviations between TOT_CYC as listed in tables 18, 19; deviations between TOT_INS as listed in tables 18, 19.

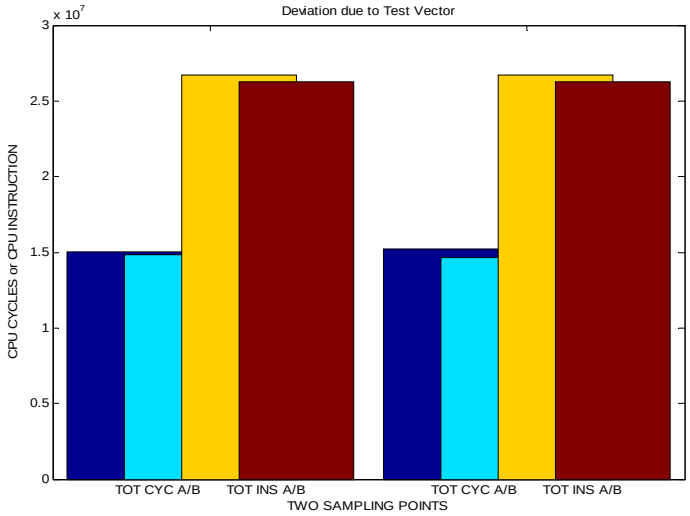

Figure 33. Real-time Samplings, EF in NSS, Test Vector Type A vs. Type B

Formulation

Table 20 lists the efficiency formula for NSS Exponentiation Function with the unit of measurements being PD, Point-Doubling, PA, Point-Adding, and Overhead Factor (OHF). Overhead Factor includes all runtime factors.

Table 20. Formulation of NSS Exponentiation Function

Unit Under Test	Formula	Unit of Measurement
NSS Exponentiation Function	NSS_EF = 524(PD) + 131(PA) + OHF (Value of OHF will be determined in the next section)	PD, PA, OHF

This test sequence shows that PEPMA computation does depend on value of k. Given k = (00100011) in binary to process PEPMA, the NSS Exponentiation Function always extracts the index for the PRE-COMP table from the leftmost four bits. Thus, k must be shifted left 4 bits for the next 4-bit extraction (This shifting also gives a name "right-to-left" exponentiation function). Furthermore, let the base-point affine vector be (x, y); then the NSS computing sequence of $k\times(x, y)$ occurs exactly as follows:

Table 21. Sequence of NSS Exponentiation Function

Iteration	**Parameter**	**Value**	**Comment**
Entry	EF shifting method	Right-to-Left	
	Affine coordinate to multiply	(x, y)	
	k	0010,0011 (binary)	Lower 8 bits Upper (521−8) = 513 bits are all zeros
	Affine-to-Projective Transformation (APT)	(RX, RY, 0)	
1	Coordinates before doubling	(RX, RY, 0)	
	Coordinates after point-doubling	(RX, RY, RZ) = (RX, RY, 0)	
	4 bits extracted from k	0010	
	Index to PRE-COMP table	2 (decimal)	4 bits extracted from k
	Coordinates from PRE-COMP table	$2\times(x, y)$ $\triangleq$ $(x2, y2)$	Affine coordinates
1.6	Coordinates after point-adding	(RX, RY, RZ) = (RX, RY, RZ) + (X = $x2$, Y = $y2$, Z=1)	Another Affine-to-Projective Transformation
2	Coordinates before doubling	(RX, RY, RZ), same as above	
	Coordinates after four point-doubling operations	16 × (RX, RY, RZ)	Exponentiation of (RX, RY, RZ) by 4
	4 bits extracted from k	0011	
	Index to PRE-COMP table	3 (decimal)	
	Coordinates after point-adding	16 × (RX, RY, RZ) + $3\times(x, y,$ RZ)	Mixed-coordinate point-adding
Exit	$k \times (x, y) = 23 \times (x, y)$	Same as above	

Analysis of NSS Exponentiation Function

The Exponentiation Function in NSS can be more efficient if NSS did not use step 1.6 (as shown in Table 21) but instead followed the recommendation from Ryabko et al. (2005) or Salomon (2006). This improvement in NSS Exponentiation Function can be further verified by examining the following operational sequence:

After processing the Affine-to-Projective Transformation, APT, the affine input coordinates (*x*, *y*) have been converted to projective coordinates (RX, RY, 0). These coordinates are the representation of an infinity point in the projective domain of input (*x*, *y*).

In the EF computation loop ③, the first iteration of the loop is special; thus, a comprehensive explanation is in order. At the beginning of the EF computation loop, the result vector (RX, RY, RZ) is set to an infinity point (RX, RY, 0). This setting of the infinity point always makes the result of point-doubling of (RX, RY, 0) to be an infinity point since a multiplication of any scalar values with an infinity point always results in an infinity point.

When this infinity point (RX, RY, 0) reaches the point-adding function for the first time (first iteration in the loop) as shown in Figure 30, the point-adding function detects the "point at infinity" condition (RZ = = 0) and returns a result (PX, PY, RZ = 1) without any further computation. This operation (PX, PY, RZ = 1) sets the Z coordinate to 1 for the first time in the EF iteration loop ③, and the result vector (RX, RY, RZ) is set to (PX, PY, RZ = 1). Note that PX, PY are the coordinates extracted from the PRE-COMP table according to the 4 bits that extracted from *k*.

When loop ③ goes into the second iteration, the point-doubling function computes the doubling of vector (RX, RY, RZ) recursively four times:

$$2 \times (2 \times(2 \times (2 \times (RX, RY, RZ)))) \triangleq 16 \times RX, RY, RZ$$

and the results are set back to result vector (RX, RY, RZ). The point-adding function then adds this result vector (RX, RY, RZ) with the next vector extracted from the PRE-COMP table.

Another interesting observation from Figure 31 and Table 22 is that if the size of extracting window were 5 bits instead of 4 bits, higher efficiency can be achieved. New values are the results of minor optimization in the implementation of EF as shown in Table 22. Units of measurement are the same as before, PD and PA.

Table 22. Alternate Formulation of NSS Exponentiation Function

Unit Under Test	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
NSS Exponentiation Function (5-bit Window)		520		PD
NSS Exponentiation Function (5-bit Window)		104		PA

Verification

Table 23 lists the efficiency formula found by BOCHS for NSS Exponentiation Function with the units of measurement being MULq, integer multiplication, and MOVq, moving quad words (64 bits). Test vectors are of type A (see Appendix O).

Table 23. Finding of NSS Exponentiation Function by BOCHS

Unit Under Test	MIN	TYP	MAX	Unit of Measurement
NSS EF	1,461,962		1,461,962	MULq
NSS EF	2,744,501		2,744,501	MOVq
NSS_OHF		To be determined when NSS_PA, NSS_PD has been derived		MULq, MOVq and other Instructions

The data collected in Table 23 is quite revealing in several ways. Since BOCHS can read exactly the number of MULq or MOVq instructions used in NSS Exponentiation Function, data in Table 23 could answer which metrics should be used to evaluate PEPMA's efficiency. Second, since the counting of MOVq instruction did exceed the counting of MULq instruction, data in Table 23 also uncovered the notion of whether the performance of PEPMA might be unknown based on existing theoretical work (for example, using only metric M, Multiplication).

Data from this table can be compared with the data in Table 32, which shows the difference in efficiency between NSS and OpenSSL PEPMA.

Finding of NSS Point-Doubling

Network Security Services implemented a software function point-doubling $R(X_3, Y_3, Z_3) = 2 \times P(X, Y, Z)$ using weighted projective transformation as described by (Cohen et al., 1998). Executing NSS Point-Doubling (PD) codes requires 4M+4S+5A+4Su+1Sh operations, where the arithmetic operators are designated as M=Multiplying, S=Squaring, A=Addition, Su=Subtraction, and Sh=Shift. These measurement units have been directly converted to the lowest measurement units MULq and MOVq using BOCHS. Table 24 recorded this operation.

Table 24. Finding of NSS Point-Doubling by BOCHS

Unit Under Test	MIN	TYP	MAX	Unit of Measurement
NSS_PD	1550			MULq
NSS_PD	3084			MOVq

Formulation

Table 25 below lists the efficiency formula for NSS Point-Doubling with the units of measurement being MULq and MOVq. The formula for NSS_PD can be constructed from values in Table 24.

Table 25. Formulation of NSS Point-Doubling

Unit Under Test	Formula	Unit of Measurement
NSS_PD	NSS_PD ≈ 1550(MULq) + 3084(MOVq)	MULq, MOVq

Analysis of NSS Point-Doubling

The data gathered in Tables 24 and 25 suggested that the target CPU had spent more time moving data than doing multiplication in PD function. The partial efficiency formulation of Exponentiation Function can now be derived as follows (OHF = Overhead Factor):

$$NSS\_EF = 524(1550(MULq) + 3084(MOVq)) + 131(PA) + OHF$$

Finding of NSS Point-Adding

Network Security Services implemented a software function point-adding of point P_1 and P_2 using weighted projective transformation as described by (Brown et al., 2001). NSS actually executed a total of 8M+3S+2A+5Su. These measurement units have been directly converted to the lowest measurement units MULq and MOVq using BOCHS.

Table 26. Finding of NSS Point-Adding by BOCHS

Unit Under Test	MIN	TYP	MAX	Unit of Measurement
NSS_PA	1808			MULq
NSS_PA	3592			MOVq

Formulation

Table 27 below lists the efficiency formula for NSS Point-Adding with the unit of measurements being MULq and MOVq. The formula for NSS_PA was constructed from the values in Table 26.

Table 27. Formulation of NSS Point-Adding

Unit Under Test	Formula	Unit of Measurement
NSS_PA	NSS_PA ≈ 1808(MULq) + 3592(MOVq)	MULq, MOVq

Analysis of NSS Point-Adding

Similar to the characteristic of NSS_PD metric, the data gathered in tables 26 and 27 suggested that in Point-Adding function, the target CPU did spend more time moving data than doing multiplication. Partial efficiency metric of Exponentiation Function now can be calculated using NSS_PA metric (OHF = Overhead Factor):

$$NSS\_EF = 524(PD) + 131(1808(MULq) + 3592(MOVq)) + OHF$$

Analysis of NSS Exponentiation Function, Revisited

The results of NSS_PD and NSS_PA, as shown in tables 25 and 27, indicate that the NSS Overhead Factor, NSS_OHF, now can be derived. Since the NSS_EF formulation was constructed earlier as NSS_EF = 524(PD) + 131(PA) + OHF, then

NSS_EF = 524(1550(MULq)+3084(MOVq))+131(1808(MULq)+3592(MOVq)) + OHF

Equivalently, NSS_EF = 1,049,048(MULq) + 2,086,568(MOVq) + OHF

From the findings earlier, the absolute computing cost for doing NSS_EF was:

Table 28. Formulation of NSS Exponentiation Function by BOCHS

Unit Under Test	MIN	TYP	MAX	Unit of Measurement
NSS EF	1,461,962		1,461,962	MULq
NSS EF	2,744,501		2,744,501	MOVq

If an approximate value of NSS_EF is 1,461,962(MULq) + 2,744,501(MOVq),

and given NSS_EF = 1,049,048(MULq) + 2,086,568(MOVq) + OHF, then the value of Overhead Factor (OHF) for NSS must exactly equal to:

NSS_OHF = 412,914(MULq) + 657,933(MOVq) + OHF

The formulation for NSS Exponentiation Function (EF) now can be compiled from the findings listed above along with the Overhead Factor, NSS_OHF.

Table 29. Formulation of NSS Exponentiation Function, Complete

Unit Under Test	Formula	Unit of Measurement
PD	NSS_PD ≈ 1550(MULq) + 3084(MOVq)	MULq, MOVq
PA	NSS_PA ≈ 1808(MULq) + 3592(MOVq)	MULq, MOVq
OHF	NSS_OHF ≈ 412,914(MULq) + 657,933(MOVq)	MULq, MOVq
NSS EF	NSS_EF≈524(NSS_PD)+131(NSS_PA)+NSS_OHF (NSS_OHF = Overhead Factor)	PD, PA, OHF

Finding of OpenSSL Exponentiation Function

In a release of open-source version 1.0.1e, dated 11 Feb 2013, OpenSSL applied a 5-bit window on the scalar *k* in PEPMA's exponentiation service shown as a computation loop ③ in Figure 34 below.

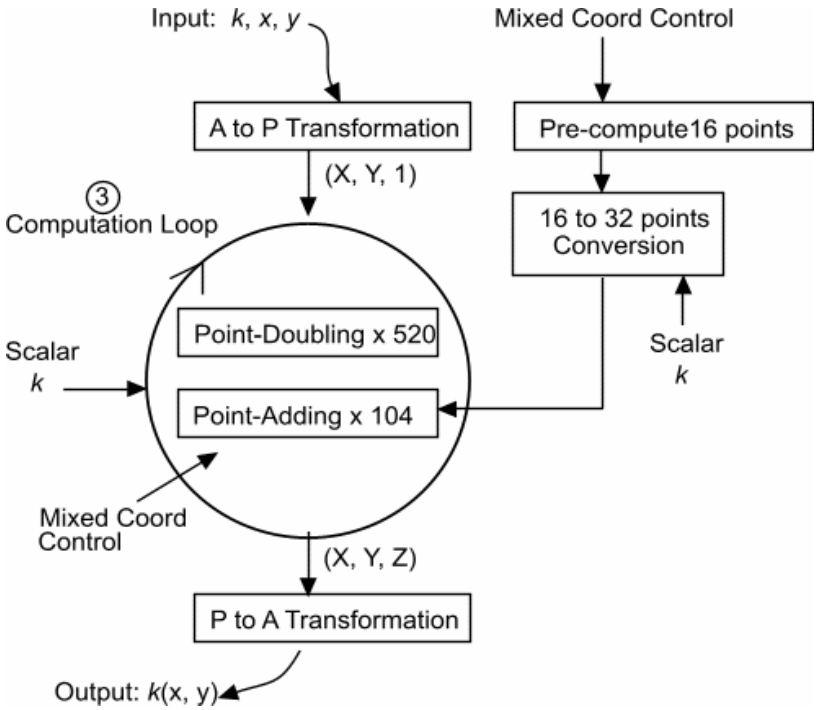

Figure 34. 5-bit Windowing Exponentiation Service in OpenSSL

Real-time Samplings with PAPI, Function EF in OpenSSL

The real-time sampling was taken from a particular desktop PC type A (see Appendix R) with test vectors type A (see Appendix O).

Table 30. Real-time Samplings, Function EF in OpenSSL, Vectors Type A

Iteration	TOT_CYC	TOT_INS	Deviation of TOT_CYC from Minimum
1	2,938,817	3,814,105	13695
2	2,935,684	3,814,104	10562
3	2,932,579	3,814,102	7457
4	2,925,122	3,814,104	0
5	2,925,122	3,814,104	0

Figure below illustrates the deviations from iteration 4, 5 of TOT_CYC as listed in Table 30.

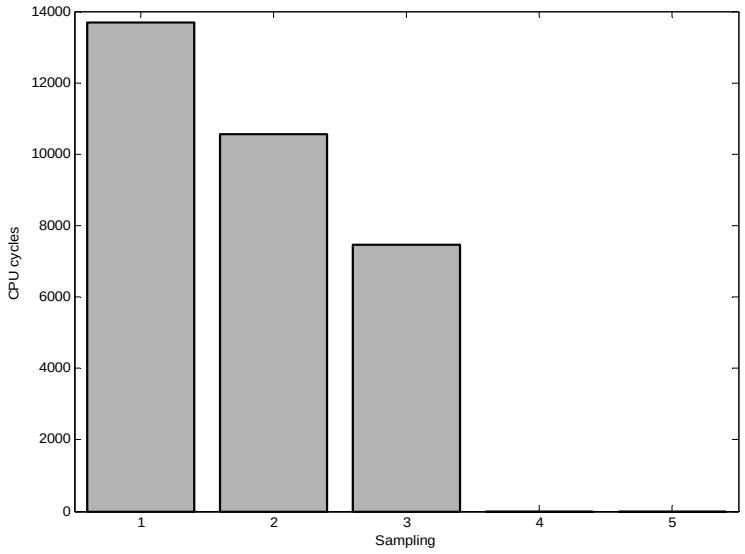

Figure 35. Real-time Samplings, EF in OpenSSL, Test Vector Type A

Another real-time sampling was taken from the same desktop PC with test vectors type B (see Appendix P).

Table 31. Real-time Samplings, Function EF in OpenSSL, Vectors Type B

Iteration	**TOT_CYC**	**TOT_INS**	**Deviation of TOT_CYC from Minimum**
1	2,944,852	3,814,106	5675
2	2,942,158	3,814,106	2981
3	2,942,373	3,814,105	3196
4	2,939,177	3,814,107	0
5	2,951,294	3,814,106	12117
6	2,845,369 *	3,814,106 *	

* Iteration 6 measured the TOT_CYC and TOT_INS parameters from a less activity runtime environment. The TOT_INS value stayed the same, but the TOT_CYC value has reduced to a smaller number. This signified a runtime dependency for TOT_CYC parameter (see Appendix R, Figure 48 for the CPU loading condition).

Figure below illustrates the deviations of TOT_CYC between Table 30 and Table 31, and deviations of TOT_INS between Table 30 and Table 31.

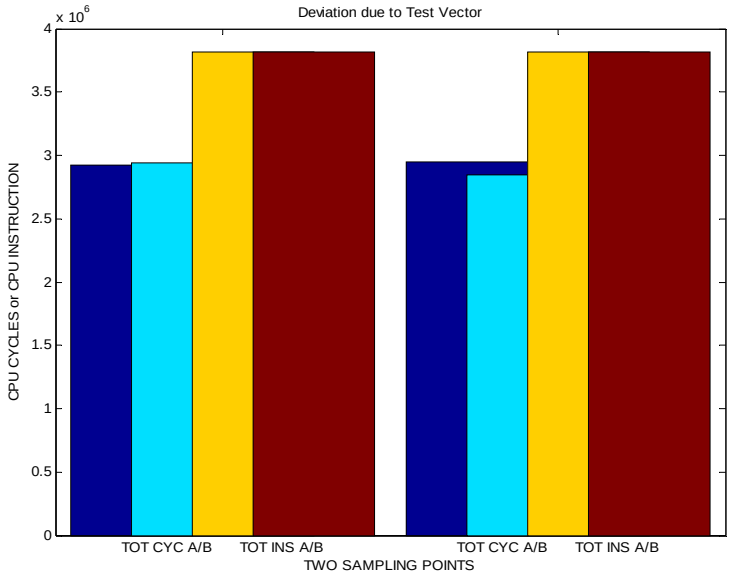

Figure 36. Real-time samplings, EF in OpenSSL, test Vector type B

Formulation

OpenSSL PEPMA makes 520 calls to the point-doubling and 104 calls to the point-adding function (OpenSSL-2, 2014). Readers are referred to Appendix H for examining the exact number of calls to PD or PA. Open_SSL Overhead part is designated as Overhead Factor (OHF) in the formula.

Table 32. Formulation of OpenSSL Exponentiation Function

Unit Under Test	**Formula**	**Unit of Measurement**
OpenSSL Exponentiation Function	OpenSSL_EF = 520(PD) + 104(PA) + OHF	PD, PA, OHF

The following three tables list the results of EF, PD, PA by BOCHS:

Table 33. Finding of OpenSSL Exponentiation Function by BOCHS

Unit Under Test	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
OpenSSL_EF	449,033			MULq
OpenSSL_EF	470,443			MOVq

Table 34. Finding of OpenSSL Point-Doubling by BOCHS

Unit Under Test	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
OpenSSL_PD	455			MULq
OpenSSL_PD	428			MOVq

Table 35. Finding of OpenSSL Point-Adding by BOCHS

Unit Under Test	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
OpenSSL_PA	1114			MULq
OpenSSL_PA	843			MOVq

From tables 33, 34, and 35 above, the cost formulas of OpenSSL EF and Overhead Factor (OHF) now can be derived:

Table 36. Formulation of OpenSSL Exponentiation Function, Complete

Unit Under Test	**Formula**	**Unit of Measurement**
PD	OpenSSL_PD ≈ 455(MULq) + 428(MOVq)	MULq, MOVq
PA	OpenSSL_PA ≈ 1114(MULq) + 843(MOVq)	MULq, MOVq
EF	OpenSSL_EF = 352,516(MULq) + 310,232(MOVq) + OHF	MULq, MOVq
OHF	OpenSSL_OHF ≈ 96,517(MULq) + 160,211(MOVq)	MULq, MOVq
OpenSSL EF	OpenSSL_EF ≈ 520(OpenSSL_PD)+104(OpenSSL_PA) +OpenSSL_OHF (OpenSSL_OHF = Overhead Factor)	PD, PA, OHF

Analysis of NSS vs. OpenSSL Exponentiation Function

From the data in Table 29 and Table 36, it is apparent that the length of computing time for NSS was longer than OpenSSL in a magnitude of at least 4 to 1.

$$NSS\_EF \approx 1{,}461{,}962(MULq) + 2{,}744{,}501(MOVq)$$

$$Open\_SSL\_EF \approx 449{,}033(MULq) + 470{,}443(MOVq)$$

This 4:1 computing ratio was not correctly shown in the theoretical work of (Brown et al., 2001), or (Cohen et al., 1998), or any other publications found in the literature review. Instead, if one summarizes the total arithmetic expenditures in the exponentiation function, he would find them to be 3668M + 3668S per (Cohen et al., 1998; Brown et al., 2001); and 2983M + 3275S per Bernstein's explicit formulation (EFD_Double, 2001; EFD_Add, 2007). From these explicit formulations, two metrics multiplications (M) and squarings (S) are the main coefficients of the cost equation to measure the performance of elliptic-curve point-multiplication *k*P residing in projective domain. Using these metrics, the cost ratios between NSS and OpenSSL would be far off as compared to the ones derived from BOCHS, or PAPI, or even from commonly used *clock*() function. The comparisons between explicit formulation, BOCHS, PAPI, and *Clock*() are summarized in Table below. The model of computing platform was of type A (see Appendix R).

Table 37. Analysis of OpenSSL vs. NSS Exponentiation Function, Test Vector A

Evaluation Method Used for Unit Under Test	**NSS**	**OpenSSL**	**Cost Ratio NSS:OpenSSL**
Explicit Formulation Metric M	3668	2983	1.25:1
Explicit Formulation Metric S	3668	3275	1.12:1
BOCHS, MULq	1,461,962	449,033	3.25:1
BOCHS, MOVq	2,744,501	470,443	5.83:1
PAPI, Number of Instructions	26,710,515	3,814,259	7:1
PAPI, Number of Clocks	15,230,372	2,939,649	5.18:1
Clock(), absolute time in mili Seconds	5101	909	5.6:1

By PAPI, the number of instructions and the number of cycles are found by executing OpenSSL PEPMA under host OS.

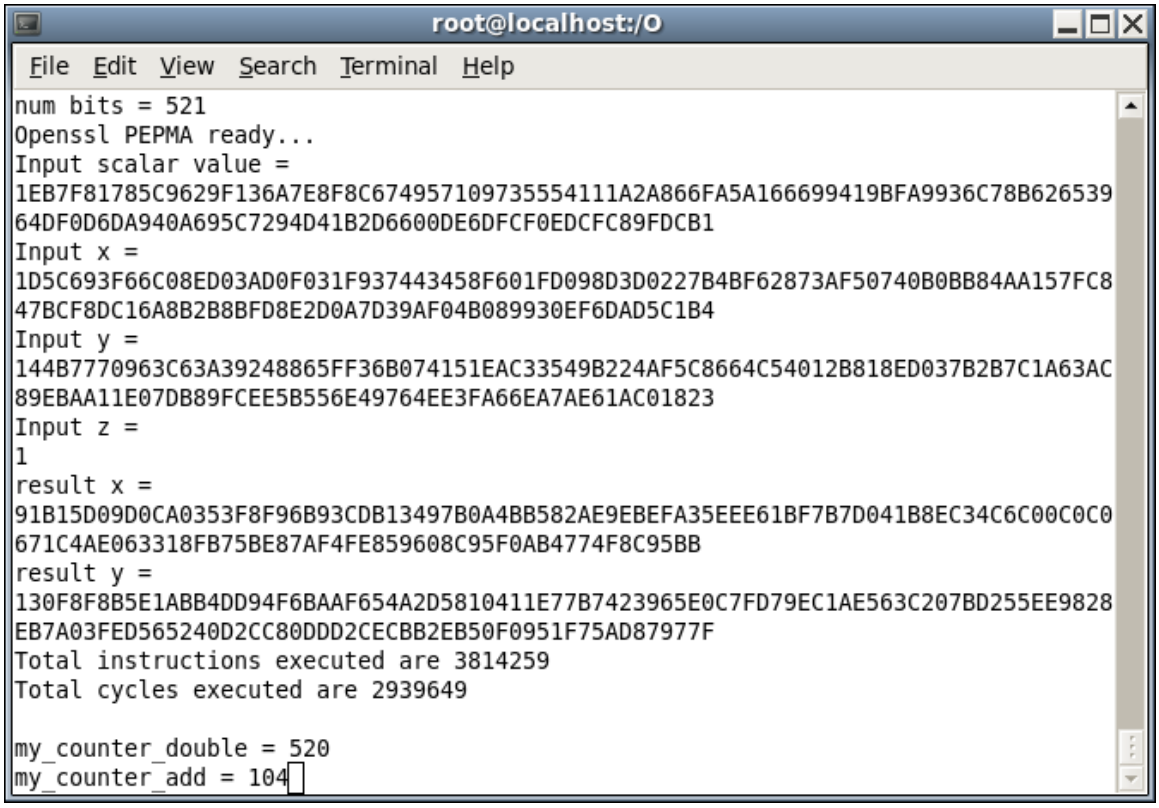

Figure 37. Result of 5-bit Windowing Exponentiation Service in OpenSSL

The outputs shown above are the results of computation with specific input test vectors as follows:

k=1EB7F81785C9629F136A7E8F8C674957109735554111A2A866FA5A166699419BFA9936C78B62653964DF0D6DA940A695C7294D41B2D6600DE6DFCF0EDCFC89FDCB1

x=1D5C693F66C08ED03AD0F031F937443458F601FD098D3D0227B4BF62873AF50740B0BB84AA157FC847BCF8DC16A8B2B8BFD8E2D0A7D39AF04B089930EF6DAD5C1B4

y=144B7770963C63A39248865FF36B074151EAC33549B224AF5C8664C54012B818ED037B2B7C1A63AC89EBAA11E07DB89FCEE5B556E49764EE3FA66EA7AE61AC01823

and the modulus for modulo arithmetic is:

m=1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFF

The unrealistic calculation from existing theoretical work has happened due to the fact that OpenSSL did apply 58-bit on multi-digit number whilst NSS used half-digit (32-bit) representation (See System Architecture, section 1.2.1, or Research in Numeric Presentation and Computation, section 2.5). Table 36 showing the overhead factor in doing EF for OpenSSL was also lower than for NSS, but it is still a significant computing cost contributing to the overall efficiency equation.

NSS_OHF ≈ 412,914(MULq) + 657,933(MOVq)

OpenSSL_OHF ≈ 96,517(MULq) + 160,211(MOVq)

None of these found differences were possible without the constructed formulations and verifications from BOCHS/PAPI. The improvement of efficiency for PEPMA can be effectively located by examining those comparative formulations and the findings presented throughout this section. The results show that there are effective ways to improve PEPMA’s efficiency based on these empirical comparisons. The findings and analyses also have shown to account for "data/memory move" metrics to candidly evaluate PEPMA's efficiency.

Finding of NSS Pre-computation

The Pre-computation is an additional cost to perform exponentiation function since the PRE-COMP table must be computed before entering EF. The performance evaluation must account for this cost to gain more accuracy. During the processing of exponentiation function, one of the significant costs is to compute the sub-exponentiation function b^e, where b is a number of bits w (window width) extracted from scalar k, and the exponent e is any small positive integer (0...15 etc.) The most common method for computing the sub-exponentiation function b^e is the sliding window approach, which enhances the efficiency at the expense of pre-computation efforts. As shown in the Methodology chapter, Figure 18 provides an idea of the sliding-window: The Network Security Services (NSS, 2013) applied a 4-bit sliding-window on the scalar k in PEPMA's exponentiation service. Additionally, as shown in Figure 20 and also in the Methodology chapter, OpenSSL applied a 5-bit window on the scalar k. However, Figure 18, Figure 20 and the associated information presented in the methodology were just a preliminary investigation which contained incomplete/undefined data. This section recorded the findings of Exponentiation Function (EF) and provided descriptions/explanations of the differences between preliminary investigation and findings of this function.

Table 38. Finding of NSS Pre-computation

Parameters	**NSS Preliminary**	**NSS in version 3.12.4**
Window Width	4-bit	4-bit
Shift Direction	Right-to-left	Right-to-left
Pre-computation of Elliptic-curve point	16 Note 1	16 Note 2

Analysis of Pre-computation

NSS Pre-computation:

Note 1: Out of sixteen Elliptic-curve points, two Elliptic-curve points do not need the computation: Elliptic-curve point zero and Elliptic-curve point P itself. Effectively, there were only fourteen Elliptic-curve coordinates (X, Y, Z), or 14 × 3 = 42 coordinates (coordinates are multi-digit-numbers) to be computed since an Elliptic-curve point in the projective domain has three coordinates (X, Y, Z).

Note 2: Same as above.

The NSS 4-bit exponentiation windowing requires a pre-computing of 15 Elliptic-curve points (pre = before entering exponentiation loop). The 15-point pre-computation calls point-doubling (PD) or point-adding (PA) services to calculate $k(x, y)$ using $k = 2$ to 15, and the coordinates (x, y) are the base coordinates of the cyclic subgroup of the chosen Elliptic curve. When $k = 1$, the pre-comp coordinates are actually the base point itself; thus, it requires no computation, just storing the coordinates in the table PRE-COMP.

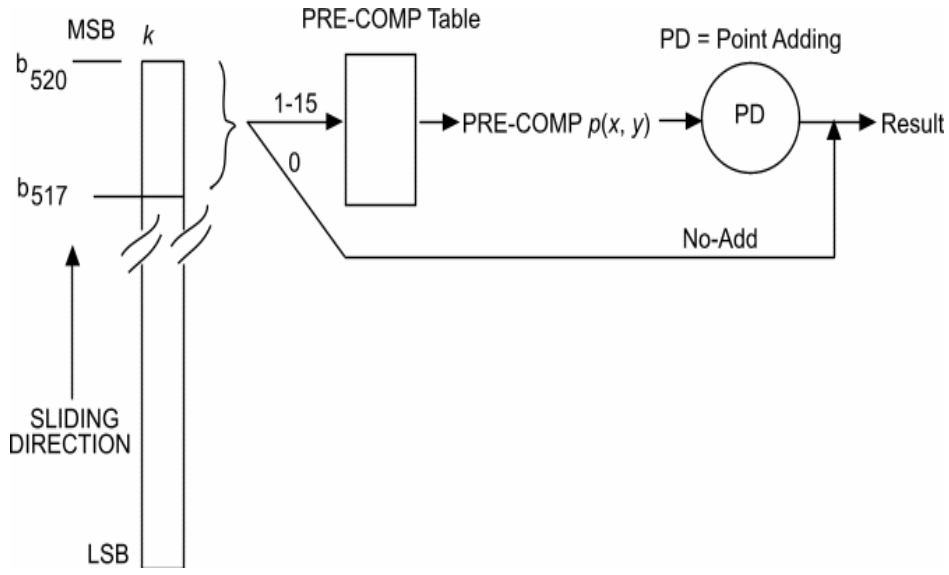

Figure 38. 4-bit Pre-comp Indexing Method used in NSS

During the exponentiation computation in the loop, *k* slides from right to left (bottom to top as shown) and 4 bits are extracted for indexing into the PRE-COMP table. The PRE-COMP value $p(x, y)$ will be used for point-adding if the index is non-zero (1...15); otherwise, a zero-value table index will signify a "No-Add" condition. The 15-point, pre-computing function makes service calls to 1 point doubling and 13 point-adding functions to completely fill the 15-point recomputed table.

Table 39. NSS Pre-computation Values in PRE-COMP Table

Table Index	***x* coordinate (Affine)**	**y coordinate (Affine)**	**Comment**
0	0	0	Infinity Point in Affine No-Add condition
1	$p(x)$	$p(y)$	Base point
2	$2p(x)$	$2p(y)$	Doubling of (x, y)
3	$3p(x)$	$3p(y)$	
4	$4p(x)$	$4p(y)$	Doubling(Doubling of (x, y))
5	$5p(x)$	$5p(y)$	
6	$6p(x)$	$6p(y)$	
7	$7p(x)$	$7p(y)$	
8	$8p(x)$	$8p(y)$	Doubling(Doubling(Doubling of (x, y)))
9	$9p(x)$	$9p(y)$	
10	$10p(x)$	$10p(y)$	
11	$11p(x)$	$11p(y)$	
12	$12p(x)$	$12p(y)$	
13	$13p(x)$	$13p(y)$	
14	$14p(x)$	$14p(y)$	
15	$15p(x)$	$15p(y)$	

Data in Table 39 uncovered that an improvement to NSS implementation can be achieved by doing three point-doublings (at table index 2, 4 and 8) and eleven point-addings.

Building the pre-computed table is done outside the computation loop. The point-adding function then uses the 4-bit window taken from *k* to index into the table without the need to call point-adding four times. This reduces calling the point-adding function by 4:1 ($131 \times 4 = 524$).

Table 40. Finding of OpenSSL Pre-computation

Parameters	**OpenSSL Preliminary (Ver 1.0.1)**	**OpenSSL in version 2.0.5**
Window Width	5-bit	5-bit
Shift Direction	Right-to-left	Right-to-left
Pre-computation of Elliptic-curve point	32 Note 3	16 Note 4

Note 3: The description of this value was incomplete in the preliminary investigation. The pre-computation of Elliptic-curve points should have been sixteen. Negation of these coordinates (X, Y, Z) makes thirty two vectors. NSS did not use this method.

Note 4: In performing the computation of exponentiation function where the negation is relatively easy, the binary signed representation (using +1, −1, 0) is meaningful because this method can decrease the amount of required pre-computation. The best signed representation is Non-Adjacent-Form (NAF), where the term "non-adjacent" implies there will not be any two bits "1" located right next to each other (0110 is not a NAF, etc.) As a result, the required pre-computation routines are reduced in half because the negative number is just a sign-changing (negation) of the positive number. This bit encoding enhances the efficiency of pre-computation since the pre-comp table now has only half of it. Even though OpenSSL used 5-bit sliding windows for computation of Exponentiation Function, there were only sixteen pre-computed values since OpenSSL implementation applied the binary signed representation as described above. The 14-point pre-computing function makes service calls to 1 point doubling and 13 point-adding functions to fill the 16-point pre-computed table. The point-adding function then uses the 5-bit window taken from k to index the table without the need to execute point-adding five times. This reduces calling the point-adding function by 5:1 ($104 \times 5 = 520$).

Finding of NSS/OpenSSL Projective to Affine Transformation

The Projective to Affine Transformation (PAT) is the last step to be executed inside an Exponentiation Function. The PAT procedure converts a final computation of Elliptic-curve coordinates (X, Y, Z) in projective domain back into the Cartesian coordinates. The results are two affine coordinates (x, y). Afterward, this conversion completes the scalar multiplication function $k(x, y)$ and returns the two affine values (x, y) to the caller of EF function. The concept of PAT is shown in Figure 4, in (NSS PEPMA, 2013; OpenSSL PEPMA, 2013; Cohen et al., 2006), and in "Concept of Point Computation in Projective Domain" of the literature reviews chapter. Table 41 shows the real-time costs for computing PAT in NSS/OpenSSL with vectors type A (see Appendix O).

Table 41. Real-time Samplings, Function PAT in NSS/OpenSSL, Vectors Type A

Iteration	NSS PAT TOT_CYC	OpenSSL PAT TOT_CYC	NSS PAT TOT_INS	OpenSSL PAT TOT_INS
1	164,904	208,209	285,500	306,445
2	163,525	206,878	285,500	306,445
3	163,631	207,377	285,500	306,445
4	163,281	209,220	285,500	306,445
5	162,703	209,184	285,500	306,445

Analysis of NSS/OpenSSL Projective to Affine Transformation

The difference of computational cost between two Projective to Affine Transformations, NSS and OpenSSL was not significant. Given that both NSS and OpenSSL must execute an inversion in PAT procedure, the results in Table 41 suggested that the computing cycles were mostly consumed by the 521-bit inversion routine. Since both NSS and OpenSSL PEPMA calculate the inversion of a number in spatial domain using an extended Euclidean algorithm (Hankerson, 2004, p. 39), the results suggested that further improvement for efficiency could not be done easily.

Finding of the Compliance Metric

The Compliance Metric (CM) of NSS PEPMA or OpenSSL PEPMA measures the compliance of computing modules to the Federal Information Processing Standard, FIPS-140-2 (FIPS-140-2, 2001). According to the records from the Cryptographic Algorithm Verification Program for certifying NSS/OpenSSL cryptographic modules (CAVP NSS, 2010; CAVP OpenSSL, 2012), both NSS and OpenSSL have received a variety of FIPS-140-2, security level 1, 2 and level 3 certifications.

Table 42. Finding of NSS/OpenSSL Compliance Metric, Level 1

Level Description	**NSS**	**OpenSSL**
Overall Complied to FIPS-140-2 Security Level 1	Validation date: 12/28/2010 Software Version: 3.12.4	Validation date: 06/27/2012;07/09/2012;07/18/2012; 10/24/2012;01/22/2013;02/06/2013; 02/22/2013;02/28/2013;03/28/2013; 05/16/2013;06/14/2013;08/16/2013; 08/23/2013;11/08/2013;12/20/2013; 06/27/2014;07/03/2014 Software Version: 2.0, 2.0.1, 2.0.2, 2.0.3, 2.0.4, 2.0.5, 2.0.6, 2.0.7
PEPMA Complied to FIPS-140-2 Security Level 1	Validation date: 12/28/2010 Software Version: 3.12.4 ECDH, ECDSA Cert. # 1280	Validation date: 07/03/2014 Software Version: 2.0.7 Module Elliptic-curve Diffie-Hellman Cert #1747

NSS Certification

The NSS software cryptographic modules have been validated five times on 08/29/1997, 1999, 2002, 2007, and 12/28/2010 (certificate #1280 including ECDH module) for conformance to FIPS-140-1 and FIPS-140-2 at security levels 1 and 2. Additionally, NSS was the first open source cryptographic library to receive FIPS-140 validation.

OpenSSL Certification

The OpenSSL version v2.0.7 has been validated on 07/03/2014, and the passing grades were recorded in FIPS 140-2 certificate #1747 (FIPS-1747, 2014). Although the software library version v2.0.7 is compatible with previous OpenSSL libraries (including versions 2.0, 2.0.1, 2.0.2, 2.0.3, 2.0.4, 2.0.5, and 2.0.6), it is important to note that the FIPS 140-1 or FIPS 140-2 certificate applies only to the version that was submitted for validation.

Formulation

The compliance metric for either NSS or OpenSSL is a complex matter, and the interpretation of these metrics might be subjective; thus, a quantitative verification using BOCHS or PAPI is not applicable for this unit-under-test. To simplify the research, the result of compliance is set to either "true" if PEPMA's technical risk assessment was low, or set to "false" if PEPMA's technical risk assessment was high. There is no comparison of compliance between NSS and OpenSSL. Nonetheless, the findings and formulations listed here still form the basis of a formal performance evaluation approach. Based on such a formal process to verify compliance, a more concrete closing conclusion about the performance may be drawn.

For the purpose of evaluating NIST 521-bit prime-field PEPMA, the Compliance Metric (CM) result was quantized to "low" from three available levels: low, moderate, and high according to the DoD source-selection procedure (Source-Selection, 2011).

Table 43. Formulation for Compliance Metric

Unit Under Test	Formula	Comment
NSS CM	CM=low	NSS has complied with FIPS. It has little potential to cause degradation of

		performance
OpenSSL CM	CM=low	OpenSSL has complied with FIPS. It has little potential to cause degradation of performance

Analysis of Compliance Metric

Since the judgment to determine the compliance is subjective, we discuss the rationale of the judgment to accept the compliance of a product in a specific military case study; and then apply the same evaluation method to this research. Readers should bear in mind that the analyses presented here are only a representation, which should provide somewhat meaningful evaluation results in a cryptographic application. With that said, the following process used by the US Department of Defense (DoD) could describe the compliance scenario for cryptographic module like PEPMA:

The DoD often solicited a Request for Proposal (RFP) publicly to fulfill an operational product requirement, after which the military procurement authorities normally follow a formal source-selection process to analyze/judge the proposals made by potential providers. As stated in the DoD source-selection procedures (Source-Selection, 2011), one of the assessments in the source-selection process is the technical risk. The term "technical," as used throughout the source-selection document, refers to all non-cost factors.

Table 44. Technical Risk Ratings

Rating	**Description**
Low	Has little potential to cause disruption of schedule, increased cost or degradation of performance. Normal contractor effort and normal Government monitoring will likely be able to overcome any difficulties.
Moderate	Can potentially cause disruption of schedule, increased cost or degradation of performance. Special contractor emphasis and close Government monitoring will likely be able to overcome difficulties.
High	Is likely to cause significant disruption of schedule, increased cost or degradation of performance. Is unlikely to overcome any difficulties, even with special contractor emphasis and close Government monitoring.

With respect to the rating of technical risk, the assessment of technical risk manifested by the identification of weaknesses, which have the potential for disruption of schedule, increased costs, degradation of performance, increased Government oversight, or the likelihood of unsuccessful contract performance. Technical risk shall be rated using the ratings listed in Table 44 (Source-Selection, 2011, p. 16).

In a specific case study, the DoD did not have any intention to evaluate/prove the validity of compliance submitted from the potential contractor. It is the potential offerer's sole responsibility to obtain certification from a third-party prior to submitting the proposal. The DoD only reviewed the “proof” of certification of technical factors and accepted it as a "passing" condition.

As advocated in the source-selection procedures, the technical factors may be divided into subfactors that represent the specific areas that are significant enough to be discriminators and to have an impact on the source-selection decision. When subfactors are used, the evaluator should establish the minimum number necessary for the evaluation of proposals. The following technical subfactos are believed to be applicable to the performance evaluation of PEPMA:

The Federal Information Processing Standards FIPS-140, entitled “Security Requirements for Cryptographic Modules,” described the government requirements for sensitive but unclassified products in terms of security and information assurance. The FIPS standards are published by the National Institute of Standards and Technology (NIST) and have been adopted by the Canadian government’s Communications Security Establishment (CSE, 2014). The security requirements for cryptographic modules also have been adopted in the financial community through the American National Standards

Institute (ANSI, 2001; ANSI, 2005). Since NSS and OpenSSL PEPMA cryptographic modules have adhered to some reasonable security requirements (i.e., implementing FIPS-approved algorithms), they are better suited for more accurate analysis than general-purpose computing systems. As such, PEPMA cryptographic with FIPS-140 ratings could provide a valuable measurement of the security controls and system information assurance in place for a given cryptographic module.

From 1994 to 2014, NIST has released three versions of FIPS-140 publications. The first version, (FIPS-140-1, 1994), was issued on 11 January 1994. This version was developed by the US government and a commercial working group and subsequently approved by the Secretary of Commerce. For security level 1, the FIPS 140-1 specification identified seven inspection areas as listed in Table 45 below.

Table 45. Compliance Metric, Seven Inspection Areas, Security Level 1

Area	**Unit Under Test**	**Comment**
1	Crypto Module	Product specification
2	Module Interface	Information flow
3	Roles & Services	Definition of module's roles and services
4	Finite-State Model	How module transitions occur
5	Software Security	Specification of the software design
6	Key Management	FIPS approved generation/distribution techniques
7	Cryptographic Algorithms	FIPS approved cryptographic algorithms for protecting unclassified information

NIST operates both the Cryptographic Algorithm Validation Program (CAVP, 2013) and the Cryptographic Module Validation Program (CMVP) where CAVP is a prerequisite to CVMP. This way, NIST ensures that cryptographic modules have been implemented correctly prior to validating their security properties. Together, these programs provide an organization with a framework to orderly certify cryptographic products against the FIPS-140 standards. Under such guidance for certification, NSS or OpenSSL applicants

have already followed at least four product requirements: (a) design a product that is compliant with the selected FIPS-140 standard, (b) prepare the documentation required for certification, (c) submit the product and documentation to an accredited testing laboratory (CAVP LABS, 2014), and (d) submit test results from the laboratory to NIST (or the Canadian government's Communications Security Establishment) for governmental approval of usage, and to receive a certification number.

In 2014, all tests under the Cryptographic Algorithm Verification Program (CAVP, 2013) are currently handled by 21 third-party laboratories that are accredited as cryptographic module testing laboratories (CAVP LABS, 2014) and by the National Voluntary Laboratory Accreditation Program. However, it is imperative to recognize that the testing laboratory could derive some of the test results from the seven inspection areas (listed in Table 45) using empirical experiments in which the results might never be fully proven. Such results can only support a passing hypothesis or can invalidate the entire validation process. Thus, while evaluating the compliance for cryptographic NSS or OpenSSL PEPMA, one could either support the passing score or reject the compliance based on seven inspection areas as shown in Table 45.

The second version, (FIPS-140-2, 2001), was issued on 25 May 2001. This version took into account changes in computing technologies and suggestions received from the communities since its first release in 1994. FIPS 140-2 defines four levels of security for cryptographic modules: security levels 1 through 4 as shown in Table 46.

Table 46. Compliance Metric, Security Level

Security Level	**Summary of Qualification**
FIPS-140-2 Level 1 Lowest security	At least one approved algorithm or approved security function used. No specific tampering detection or intrusion prevention mechanisms employed
FIPS-140-2 Level 2	Level 1 + Module must show evidence of tampering or intrusion
FIPS-140-2 Level 3	Level 2 + Module must prevent intruder from gaining access
FIPS-140-2 Level 4, highest security	Level 3 + Provide reliable level of intrusion detection and prevention system

Several security requirements pertaining to each security level have been incorporated into Version 2. This addition was the direct result of the feedbacks from the communities. The rationale for having different levels of security follows: The total number of cryptographic service modules is usually large. This is certainly true in the case of NSS or OpenSSL which has been around the industry for decades (NSS, 2013; OpenSSL, 2013). The security aspects of these modules are complex and costly for verification and validation. Thus, not all modules can be certified at once. Instead, only special FIPS object modules have been derived from un-certified core components and brought in for certification at the third-party laboratories such as (CAVP LABS, 2014). These FIPS object modules were carefully designed with specific compilation instructions so that the certification can be transferred with minimal effort to the products applying NSS/OpenSSL cryptographic service modules.

As shown in Table 42, both NSS and OpenSSL have gone through several certifying iterations and have been working well in the fields. The evidence of having inspections in seven or more areas covering Elliptic-curve Diffie-Hellman and Elliptic-curve Digital Signature Algorithm indicated that both NSS or OpenSSL PEPMA implementations are believed to be adequately stable; and that the codes can be applied to Elliptic-curve public key exchange cryptography to ensure authenticity in the public key infrastructure.

The third version, FIPS 140-3, is currently under development this year. In the first draft, NIST introduced one additional security level: information assurance (level 5) and two new power analyses to measure the signal leakages (Simple Power Analysis and Differential Power Analysis). We discuss these new security aspects with respect to the performance of PEPMA as follows:

One way to add dimension to the performance evaluation is to leverage the measurements of outliers, that is, coding practices which produced signal patterns, or use of data outside of the norm. According to Herrmann (2007) and Fenton (1996), the compliance to federal standards could detect and correct outliers and thus contribute to the overall performance of NSS/OpenSSL PEMA. Per Keyes (2005), further analysis of CM showed that any service modules that have adequately complied tend to have lower complexity and will eventually lead to better performance in the field. Thus, it is imperative to accept that these new security aspects in FIPS-140-3 might contribute to PEPMA's overall performance if one chooses to emphasize the importance of information assurance level 5. In this evaluation, we did not emphasize the importance of information assurance toward the combined key performance indicator for both NSS and OpenSSL; thus, for the design of cryptographic modules, the technical risk level is still believed to be low for all intended purposes of the performance evaluation of PEPMA.

Finding of Cyclomatic Complexity Metric

The Cyclomatic Complexity measures the structural complexity of NSS or OpenSSL PEPMA's software modules. The terminology "Static Complexity Metric" was used in the older literature (IEEE 982.1, 1988, p. 23; IEEE 982.2, 1988, p. 60); however, the term "Cyclomatic Complexity" is more commonly used today. Readers are referred to NIST Special Publication 500-235 (Watson & McCabe, 1996) for a more detailed discussion of Cyclomatic Complexity Metric.

This section is the follow-up from the previous methodology chapter, which already constructed the Static Complexity Metric (SCM) in terms of the number of edges, E; number of nodes, N; and a constant 1:

$$SCM = E - N + 1$$

Before proceeding to evaluate the Cyclomatic Complexity Metric (CCM) with respect to PEPMA, it will be necessary to adjust the constant 1, which assumed the number of exit path to be a loop-back to itself. If the number of exit path, P, is other than a loop-back, then the SCM formula becomes:

$$CCM = E - N + 2P$$

The following tables present the findings for NSS Cyclomatic Complexity Metric of Point-Doubling (PD), and Point-Adding (PA). Both PD and PA were called by two functions: Exponentiation Function and Pre-computation.

Table 47. Findings of NSS Cyclomatic Complexity Metric of PD

Coefficients of NSS Cyclomatic Complexity of Point-Doubling Function	**MIN**	**TPY**	**MAX**	**Unit of Measurement**
E = number of edges	123			Scalar
N = number of nodes	82			Scalar
P = number of exit paths	1			Scalar
CCM_NSS_PD	43			Scalar

Table 48. Findings of NSS Cyclomatic Complexity Metric of PA

Coefficients of NSS Cyclomatic Complexity of Point-Doubling Function	**MIN**	**TPY**	**MAX**	**Unit of Measurement**
E = number of edges	90			Scalar
N = number of nodes	60			Scalar
P = number of exit paths	1			Scalar
CCM_NSS_PA	32			Scalar

Formulation

The formulations of CCM were derived from the coefficients of Cyclomatic Complexity:

Table 49. NSS CCM Formulations

Sub-Module	**Formula**	**Unit of Measurement**
NSS PD	CCM = 123 - 82 + 2	E, N, P
NSS PA	CCM = 90 - 60 + 2	E, N, P

The following tables present the findings for OpenSSL Cyclomatic Complexity Metric of Point-Doubling (PD), and Point-Adding (PD). Both PD and PA were called by two functions: Exponentiation Function and Pre-computation.

Table 50. Findings of OpenSSL Cyclomatic Complexity Metric of PD

Coefficients of NSS Cyclomatic Complexity of Point-Adding Function	**MIN**	**TPY**	**MAX**	**Unit of Measurement**
E = number of edges	34			Scalar
N = number of nodes	34			Scalar
P = number of exit paths	1			Scalar
CCM_OpenSSL_PD	2			Scalar

Table 51. Findings of OpenSSL Cyclomatic Complexity Metric of PA

Coefficients of Cyclomatic Complexity	**MIN**	**TPY**	**MAX**	**Unit of Measurement**
E = number of edges	73			Scalar
N = number of nodes	71			Scalar
P = number of exit paths	1			Scalar
CCM_OpenSSL_PA	3			Scalar

Formulation

The formulations of CCM were derived from the coefficients of Cyclomatic Complexity:

Table 52. NSS CCM Formulations

Sub-Module	Formula	Unit of Measurement
OpenSSL PD	CCM = 34 - 34 + 2	E, N, P
OpenSSL PA	CCM = 73 - 71 + 2	E, N, P

Analysis of Cyclomatic Complexity Metric

The cyclomatic complexity of PEPMA source code is the counting of linearly independent paths through the service module (Watson & McCabe, 1996). A simple case example is when CCM=2. If the PEPMA source code does not have any decision branching such as an "if" statement, then the Cyclomatic Complexity Metric CCM equals to 2, since there exists only one edge (E=1), one node (N=1), and one exit path (P=1) throughout the module. However, if the PEPMA service module has an "if" statement, there will be three edges through the code: one edge where the "if" statement is evaluated as a "true" and two edges where the "if" statement is evaluated as a "false." In this case, E=3, and N equals to 2. Thus CCM = 3. Simply, the Cyclomatic Complexity Metric = (ifs + loops + cases − return + 2)

Prior studies of cyclomatic complexity have shown a correlation between a program's structural complexity and its testability. The scalar level of cyclomatic complexity suggests that a software module of higher complexity tends to produce higher probability of errors when fixing or enhancing the source code. Thus a high level of CCM denotes a service module that exhibits lower reliability, a difficulty to test, more costs to

certify, and a difficulty to maintain. Hence, a higher level of CCM can be thought of in terms of lower performance, and vice versa. The Software Engineering Institute (SEI, 1997, p. 147) established the thresholds of CCM as follows:

Table 53. CCM Level

CCM Level	**Complexity**	**Risk**
1-10	Simple Module	Not much risk
11-20	Moderate Complex Module	Moderate risk
21-50	Complex Module	High risk
51-above	Very complex, untestable	Very high risk

The comparison of coding complexity between NSS and OpenSS were made using Cyclomatic Complexity Metrics with the thresholds of CCM as shown above.

Table 54. Comparison between NSS and OpenSSL CCM

Sub-Module	**CCM_NSS**	**CCM_OpenSSL**	**Comment**
PD	43	2	NSS PD Module has higher risk
PA	32	3	NSS PA Module has higher risk
PD+PA	75	5	Used for calculating cKPI

Findings of Weighted Information Flow Complexity

The Weighted Information Flow Complexity (WIFC) measures inter-module structural complexity. The detailed characteristics of WIFC can be found in (Herrmann, 2007, p. 121; IEEE 982.2, 1988, p. 74).

$$\text{WIFC} = (\ fanin \times\ fanout)^2 \times length$$

where:

fanin = Number of sinking capability into the module (module loading)

fanout = Number of sourcing capability from the module (module supplying)

length = Number of source statements in the module

The following tables present the findings for NSS/OpenSSL Weighted Information Flow Complexity of Point-Doubling, PD, and Point-Adding module, PA. The values of *fanin* and *fanout* were derived from NSS/OpenSSL Exponentiation Function and pre-computation as the callers to PD or PA.

Table 55. Findings of NSS Information Flow Complexity of PD

Coefficients of Information Flow Complexity	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
fanin	2	2	2	Scalar
fanout		12	12	Scalar
length			58	Scalar
NSS_WIFC_PD			33408	Scalar

Table 56. Findings of NSS Information Flow Complexity of PA

Coefficients of Information Flow Complexity	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
fanin	2	2	2	Scalar
fanout		11	11	Scalar
length			45	Scalar
NSS_WIFC_PA			21780	Scalar

Table 57. Findings of OpenSSL Weighted Information Flow Complexity of PD

Coefficients of Weighted Information Flow Complexity	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
fanin	2	2	2	Scalar
fanout		14	14	Scalar
length			33	Scalar
OpenSSL_WIFC_PD			8448	Scalar

Table 58. Findings of OpenSSL Weighted Information Flow Complexity of PA

Coefficients of Weighted Information Flow Complexity	**MIN**	**TYP**	**MAX**	**Unit of Measurement**
fanin	2	2	2	Scalar
fanout		14	14	Scalar
length			68	Scalar
OpenSSL_WIFC_PA			17408	Scalar

Analysis of Weighted Information Flow Complexity (WIFC)

The high level of information flow complexity indicates a possibility for broader testing or major redesign. Additionally, the usage of WIFC might offer the following advantages: (a) controlling the service modules with improved efficiency, (b) enabling improvement in terms of complexity and flow content, and (c) more accuracy in performance comparison. In short, the WIFC is another important performance factor of PEPMA, which contributes to the overall performance evaluation.

The *fanin* coefficient of WIFC is the number of other modules calling to the unit-under-test; thus, *fanin* indicates the sinking capability. The *fanout* is the number of other modules being called by this unit-under-test; hence, it is the sourcing capability of the unit-under-test.

The high level of *fanin* indicates a better design structure of the module. A higher *fanin* level also reveals that the unit-under-test has been called heavily. The *fanin* parameter also shows the re-usability, and thus, it can help the code implementer to reduce redundancy during coding.

The *fanout* coefficient indicates the coupling between this unit-under-test and other modules in the system. A high level of *fanout* means a highly coupled module. A high level of *fanout* also indicates that the unit-under-test depends highly on the other module; thus, a high level of *fanout* indicates a poor design structure. A high level of *fanout* also increases the cost to maintain. Any code changes in the module will require modifications to the other modules and thus directly contribute to the increased level of maintenance.

Since the number of source-code statements can vary widely, the module can be very simple or very complex. This suggests that the metric WIFC is to be weighted with coefficient *length.* Readers are referred to the literature from (Herrmann, 2007, p. 121; IEEE 982.2, 1988, p. 74) for more descriptions of this parameter. A comparison between NSS and OpenSSL CCM is shown below.

Table 59. Comparison between NSS and OpenSSL WIFC

Sub-Module	**WIFC_NSS**	**WIFC_OpenSSL**	**Comment**
PD	33408	8448	NSS PD Module has higher information flow complexity
PA	21780	17408	NSS PA Module has higher information flow complexity
PD+PA	55188	25856	Use in cPKI

Finding of Module Maturity Index

The Module Maturity Index (MMI) measures the effect of changes from one software module baseline to the next. The findings of MMI were derived with two different software versions based upon a general discussion in (Herrmann, 2007, p. 121), as originated in standards (IEEE 982.1, 1988, p. 19; IEEE 982.2, 1988, p. 51), or as described in other standards (IEEE 982.1, 2005, p. 26).

$$\mathrm{MMI} = \frac{\mathrm{M} - (\mathrm{A} + \mathrm{C} + \mathrm{D})}{\mathrm{M}}$$

M = Number of modules in the baseline

A = Number of added modules from baseline

C = Number of changed modules from baseline

D = Number of deleted modules from baseline

Table 60. Module Maturity Index, NSS

MMI Coefficient	**NSS Version 3.12.4**	**NSS Version 3.16.1**
M	1758	1785
A	0	27
C	0	0
D	0	0
NSS_MMI	1	0.98

Table 61. Module Maturity Index, OpenSSL

MMI Coefficient	**OpenSSL Version FIPS 1.2.3**	**OpenSSL Version FIPS 2.0.5**
M	980	1044
A	0	64
C	0	197
D	0	389
OpenSSL_MMI	1	0.3

Analysis of Module Maturity Index

In order to derive the Module Maturity Index, MMI, a side-by-side file comparison was set out to work on files with extension *.c and *.h. With these specific settings for file filtering, WinMerge – a "file-diff" program – computed six coefficients A, C, D for NSS and OpenSSL as shown in tables 60 and 61. From these coefficients, the Module Maturity Index for NSS was found to be 0.9, and the MMI for OpenSSL was 0.3. Apparently, NSS implementation was more mature than OpenSSL implementation.

Finding of Functionality Metric

The Functionality Metric (FM) measures the interoperability between available point-doubling and point-adding functions. There were several alternate arithmetic approaches currently available to construct Point-Doubling (PD) and Point-Adding (PA) functions in a projective domain. However, the mathematical results of NSS or OpenSSL PEPMA are still the same in applying these alternate PA and PD functions. The literature (IEEE 982.2, 1988, pp. 70-71) provided a general discussion of this metric.

Table 62. Functional Metric

Module	**Description**	**Interoperable with**
Point Doubling type 1 (Cohen et al., 1998)	Used in NSS Point-Doubling function	OpenSSL PD
Point Doubling type 2 (Brown et al., 2001)	Used in OpenSSL Point-Doubling function	NSS PD
Point Adding type 1 (Brown et al., 2001)	Used in NSS Point-Adding function	OpenSSL PA
Point Adding type 2 (Brown et al., 2001)	Used in OpenSSL Point-Adding function	NSS PA

Analysis of Functional Metric

The Functional Metric indicates that Point-Doubling or Point-Adding functions ─ as suggested in (Cohen et al., 1998) or in (Brown et al., 2001) ─ are mathematically interchangeable between NSS and OpenSSL. Although there was limited evidence

showing the benefit of exchanging modules in terms of efficiency, it is interesting to note that in some applications, exchanging modules to gain federal compliance might be beneficial.

Summary of Key Performance Indicators

For the metric EMF, equation TOT_INS should include all CPU instructions. For instance, one additional coefficient RET in the equation TOT_INS will make the result of EMF accurate. This case study is described in Appendix U, and in Appendix V for a simple 64 bit multiplication:

TOT_INS = MULq + MOVq + RET

For comparison, one should convert TOT_INS to the total number of CPU cycles. However, there is always a "cost of quality" associated with measuring instrumentation and modular improvements. Intensive analysis labor for adding more coefficients into the BOCHS equations will be required to construct EMF accurately. Consequently, the TOT_CYC values approximated by PAPI were applied: 15,230,372 for NSS EF and 2,939,649 for openSSL EF. Lower PAPI value indicates a better performance. For computing the combined key performance indicator, the perform ratio between OpenSSL and NSS is 518/100. Higher value indicates a better performance.

As shown in Table 42, both NSS and OpenSSL have gone through several certifying iterations and have been working well in the fields; thus, the CM performance scores for NSS and OpenSSL are even. For computing the combined key performance indicator, the CM scores are normalized to 100.

Table 54 shows the results of code walk-through and inspection of CCM: 75 for NSS and 5 for OpenSSL. Lower values indicate a better performance. For computing the

combined key performance indicator, the perform ratio between OpenSSL and NSS is 1500/100. Higher value indicates a better performance.

Table 59 shows the results of code walk-through and inspection of WIFC: 55,188 for NSS and 25,856 for OpenSSL. Lower value indicates a better performance. For computing the combined key performance indicator, the perform ratio between OpenSSL and NSS is 213/100. Higher value indicates a better performance.

Table 60 and 61 show the MMI values as the results of a side-by-side file comparison on the source codes. The final scores are 0.98 for NSS and 0.3 for OpenSSL. For computing the combined key performance indicator, the perform ratio between OpenSSL and NSS is 33/100. Higher value indicates a better performance.

Table 62 shows the results of code walk-through and inspection of FM: the performance scores for NSS and OpenSSL are even. For computing the combined key performance indicator, the FM scores are normalized to 100.

Finding of Combined Key Performance Indicator

The combined Key Performance Indicator (cKPI) is the final single scalar-value to provide the overall performance of PEPMA. It has been shown in Herrmann (2007, pp. 123-124) that in order to derive the cKPI, the evaluator should determine the importance level of each individual performance indicator. Subsequently, the weighted factors can be derived from these importance levels. The lack of a proper approach to determine the importance level might be a handicap for a practical application; however, in this study, an observation that emerged from the findings of importance levels, and weighted factors was that EMF usually carries the most weight; but there might be some application where the Certification Metric may become a greater governing factor for the performance of

PEPMA; thus, the determination of importance levels and weighted factors has been left off from this study and should be determined on a case-by-case application. With that said, the weighed factors listed in Table 63 on the fifth column were the author's own opinions while working with Certificate and Authority in 2013.

Table 63. Final cKPI of NSS/OpenSSL PEPMA

Key Performance Indicator	**Max Value**	**NSS Score (Reference)**	**OpenSSL Score**	**Weight %**	**Subtotal (OpenSSL)**
EMF		100	100*15,230,372/ 2,939,649	60	311
CM	100	100	100	15	15
CCM		100	1500 (normalized)	10	150
WIFC		100	213 (normalized)	7	15.1
MMI		100	33	5	1.65
FM	100	100	100	3	3
cKPI		**100**		**100**	**496**

A higher cKPI value signifies a better performance as compared to NSS. Overall, OpenSSL's performance is 5 times better than NSS's performance. The method for calculating a final value of cPKI = 496 was briefly described in methodology section. Detailed industry practice and recommendations for calculating a value of cKPI can be found at these cited sources (Herrmann, 2007; Hennessy, 2006).

Chapter Summary

This chapter reports the findings from six units of analyses, associated formulations, and analysis of the findings to show evidence that the performance of PEPMA might be unknown based on existing theoretical work. More key performance indicators to evaluate PEPMA's efficiency are also presented in the findings, rather than just the three metrics (M, S and I) suggested by the existing theoretical work. The findings from two studying cases suggested that the efficiency metrics and formal verification method along with other key performance indicators (CM, WIFC, CCM, MMI, FM) can be used to accurately evaluate the performance of Projective Elliptic-curve Point Multiplication in 64-bit x86 Runtime Environment.

What has emerged in the findings and analysis of the key performance indicators is the overall performance of PEPMA, which measured by the combined key performance indicator, should be a function based on role-sharing rather than a single dedicated performance indicator. The role-sharing relates to the importance of each role, and it must be carefully determined on a case-by-case basis. Finally, based on the empirical comparison of sub-modules and low-level services, clearly that a formal performance evaluation approach will provide a useful tool to enable the code implementers to improve PEPMA's efficiency.

Chapter 5

Conclusion, Implications, Recommendations, and Summary

This chapter is organized into five sections. The beginning section titled "Objective and Goal" reiterates several main evaluation methodologies and summarizes the purpose of this study. The concept of reductionism, which is finding the most fundamental metrics and formulations and reducing them to one final result, is central to this research. In the section titled “Conclusion,” we present our thoughts regarding reductionism. The “Implications” section recapitulates the findings and the results from chapter 4. On logical grounds, there is no compelling reason to disagree with the generality of this research. Section 3 implies that this research on PEPMA can be realistically expanded beyond its original goal and scope. Section 4 “Practical Applications” provides ways to apply this study to industry applications; and the section titled “Recommendation” provides a recommendation of changes to improve PEPMA’s performance evaluation. Lastly, a "Future Work" clause briefly lists out future tasks that could enhance the performance evaluation.

Objective and Goal Review

It is becoming increasingly difficult to ignore the fact that network penetration by malicious software is getting more sophisticated every day. According to US-CERT, more than one hundred thousand damaging intrusion attacks to the U.S. military network have occurred every year. This highlights the need for the next-generation public-key exchange design to encompass high withstanding capability. This requirement poses a major challenge to software professionals who will need to search for an innovative

approach to derive longer private keys with the best performance possible. For this reason, the main topic of this study was to focus on a specific performance comparison of projective Elliptic-curve point-multiplication in a 64-bit x86 runtime environment ─ an effort to compare quantitative key performance indicators between two FIPS-certified Projective Elliptic-curve Point-Multiplication Agents for the purpose of improving PEPMA itself.

To realize such empirical comparisons, the research focused on uncovering whether the performance of PEPMA might be unknown based on existing theoretical work and revealing what metrics should be used to truthfully evaluate efficiency through the use of virtual machine and performance hardware counters. After these questions have been satisfactorily answered, the evaluator eventually will attempt to seek ways to improve PEPMA's final performance based on such empirical comparisons.

In order to fulfill these objectives, we constructed a specific performance measurement system that targeted two FIPS-certified PEPMA open-sources: NSS and OpenSSL. We used various means to extract the findings. They were found through the review of existing industrial documentation and the active contents of cryptographic certificates. They were also found by examining NSS/OpenSSL open-source codes, and by discovered the efficiency through executable-binaries that run under both host and guest Operating Systems. We were able to complete all objectives of this study successfully. The ultimate goal of this research was to develop and suggest a repeatable and deterministic evaluation approach of the performance of PEPMA.

As previously stated in the product requirements, the evaluation approach shall provide a detailed framework to construct a better evaluation method with deterministic

verification systems. Thus, the final contribution to the field of cryptography is a formal and practical evaluation method that can guide the evaluator through the performance improvement of PEPMA.

Conclusion

This study provided the following answers to the three research questions posted in Chapter 1.

Research Question 1: Is the performance of PEPMA unknown based on existing theoretical work?

Case evidence for this question showed that the performance of PEPMA is unknown based on existing theoretical work. In order to accurately describe the performance of PEPMA, the evaluator should include at least six Key Performance Indicators and combine them into a final value cKPI as listed in Table 63.

The quantity Efficiency Metric and Formulation was derived from the software reviews combined with the usage of a special virtualization technology and hardware performance counters. The computational efficiency comparison leveraged around these two technologies.

The judgment of Compliance Metric is subjective; thus, the research provided a discussion for ruling the compliance with respect to the DoD source-selection guide. The remaining key performance indicators (CCM, WIFC, MMI and FM) are quantitative metrics. They were derived from the manual software reviews. The manual software reviews in this study adhered to the code walk-through and software inspection formal process as recommended in IEEE standards.

Research Question 2: What metrics should be used to truthfully evaluate PEPMA's efficiency?

NSS/OpenSSL case evidence and data validations from BOCHS and PAPI showed that the metric to truthfully evaluate PEPMA's efficiency is the cost equations provided by the CPU instruction software counters. The CPU instruction software counters are realized with machine virtualization technology, BOCHS.

The instruments BOCHS which provided machine virtualization can accurately measure the total number of CPU instructions and CPU cycles. It can also indicate what types of CPU instruction that PEPMA uses. In short, machine virtualization allowed accurate counting each CPU instruction; and at the same time, provided an indication of what CPU instructions are being used. By analyzing these parameters in the cost equations, the evaluator will be able to determine ways to improve PEPMA’s efficiency and targeting precisely which software module can be improved, even without library source code. It is also feasible to derive an accurate cost equation by expanding the BOCHS software counters to cover all CPU instructions.

There is always a “cost of quality” associated with measuring instrumentation and improvements. Intensive analysis labor for adding more coefficients into the BOCHS equations will be required to construct this metric.

The second most accurate metric is the PAPI hardware CPU instruction counter. This quantity can be measured quickly and effortlessly but it cannot be used for modular improvement. The third accurate metric is the PAPI hardware CPU cycle counter. This quantity can also be measured quickly and effortlessly but it, too, cannot be used for

modular improvement. All PAPI type measurements are less accurate than BOCH instrumentation; however they do complement the construction of cost equations.

Research Question 3: Are there ways to improve PEPMA's efficiency based on the empirical comparison?

Documentation search and/or certificates were used to determine some possible areas of improvement with respect to PEPMA's overall performance. However, the primary method of searching for ways to improve efficiency was through the examination of NSS/OpenSSL source codes. Subsequently, the cost formulas for the empirical comparison and data validation were constructed. Furthermore, PEPMA's efficiency can also be improved by running the executable binaries under both host and guest Operating Systems (Virtual Machine using BOCHS) then comparing the results to the program outputs.

In general, the reductionism method is a powerful approach for studying and improving complex mathematical systems ─ systems such as PEPMA. This is an approach to comprehend each level of complexity in terms of the next lower level; and perhaps, this is the traditional philosophy of reductionism simply stated: "Let us find the most fundamental parts and laws." Gell-Mann (1996) and Morowitz (2002) further see the complex system that always possesses multiple complexities; and such complexities always reside scattering in an extended space of dimensionalities.

While the introduction, literature review, and methodology sections already provided some evidence to describe those complexities, the findings from six units of analyses in the results chapter have uncovered those extended spaces of dimensionalities. The exploration of findings has shown concrete facts that the performance of PEPMA

was incomplete based on existing theoretical work, which operates and resides only in a one-dimensional metric. Additionally, together with the findings, the constructed formulas and multiple data analyses have confirmed which metrics could be used to truthfully evaluate PEPMA's efficiency.

Furthermore, the findings and the results based on six units of analyses suggested a comprehensive setup for a formal evaluation method with several Key Performance Indicators instead of a single indicator as suggested through existing theoretical work. A combined Key Performance Indicator, cPKI, then can be derived from these individual Key Performance Indicators; the final single numerical score registered in the combined Key Performance Indicator will show how well a PEPMA performs relative to other PEPMA(s).

Implications

Although the ultimate goal of this research singularly focused on an evaluation approach of the performance of PEPMA, this research can be realistically expanded beyond its original goal and scope.

Practical Applications

One possible application in the cryptographic field is the code implementation of Elliptic-curve Diffie-Hellman (ECDH) public-key-exchange protocol running on limited computing-power platforms. These platforms may include tablet PCs, wrist-worn computers, or futuristic micro-size computing gadgets. Because of limited computing-power, these tablet PCs or wrist-worn computers must rely on highly efficient public-key exchange, PEPMA in particular, to accomplish its public-key exchange function in a very short duration; at the same time, the computing platforms must also meet or exceed other

performance indicators such as Cyclomatic Complexity. The results of this research support the idea that to effectively improve PEPMA, the evaluator should have a way to accurately measure it first. Along with a concept of point-computation in a projective domain, a deep understanding of how PEPMA was implemented and processed is the key to realize a high-performance Elliptic-curve Diffie-Hellman protocol.

Another possible application of this research relates to Intrusion Detection Systems (IDS). As the name implies, IDS is a device that is specifically designed to detect and prevent malicious intrusion to a system. However, before it can effectively perform that defensive task, the internal structure of IDS must provide accurate and reliable intrinsic services. The performance evaluation and measuring instruments of PEPMA may be used to improve IDS design. Furthermore, the performance evaluation of PEPMA can be mapped directly to the performance evaluation of IDS with minimum re-engineering efforts since most methodologies and verification tools have already been built.

The third benefit as the result of improving efficiency of PEPMA can be directed at the cryptographic hardware units. For instance, most Internet data traffic coming in and out of a military base must go through several layers of data filtering. These session-based digital filtering functions are being executed inside a piece of high-speed hardware known as "the Guard" which is capable of accomplishing traffic filtering at a data rate of ten or more Giga-bit per second (multiple Giga bytes per second filtering capability in real-time). To realize this lightning task, the Guard must transform all data into a projective domain, process data filtration in this domain, and convert them back into time-domain – all done in real-time on custom-made hardware fabrics (or using Field Programmable Gate Array, FPGA). The efficiency evaluation of PEPMA may be useful

during design and/or evaluation of the Guard and thus might result in better hardware design and firmware/micro code engineering.

Recommendations

Applying Formal Evaluation Approach:

What has emerged from the result of this study was the overall performance of PEPMA measured by a combined key performance indicator. This indicator suggested that the performance measurement should be a function based on role-sharing rather than a single dedicated performance indicator. This research sought to remedy the use of an insufficient one-dimensional performance indicator as suggested in theoretical work. The objective was done by reviewing other methods used in industry during a time period spanning two decades. Because of this insufficient performance merriment, one of the recommendations is to instigate a change to the way performance evaluation has been performed. The rationale behind giving out this suggestion is to remedy a problem: It is necessary but insufficient to evaluate the performance of Elliptic-curve scalar point-multiplication in projective geometry using the total number of single-digit non-modular multiplication metric, or single-digit non-modular squaring metric, or under an unspecified computing architecture.

Another recommendation relates to the use of advanced measurement instrumentation: A virtual machine can be used to precisely and accurately measure the performance metrics. As a matter of fact, commercial industry and government entity are utilizing virtual machines today to suppress adversaries on the world-wide network by accurately measuring suspicious activities occurring real-time on a piece of malware. Bottom-line, using a virtual machine to acquire metrics instead of relying on the primitive

clock() measurement method will offer consistent and accurate results on the performance comparison of a projective Elliptic-curve point-multiplication in a 64-bit x86 runtime environment.

Efficiency Improvement:

A particular feature was noted at NSS exponentiation procedure where a number of projective point-adding can be reduced by increasing the width of the sliding-window from 4 to 5. Even though the exponentiation procedure uses a 5-bit sliding-window for the computation, there will be only sixteen pre-computed values needed since the implementation could apply the binary signed representation as described previously. This reduces calling the point-adding function by 5:1 instead of 4:1 as currently implemented in NSS.

During the findings, another property was spotted in NSS half-digit 32-bit numeric representation that can be adjusted for improving efficiency. The conversion of existing codes from a half-digit 32-bit representation to a 65-bit numeric representation (64-bit with hardware carry bit) is possible in a 64-bit x86 system. Such successful conversion can significantly change the computing efficiency of PEPMA. This improvement can also be applied to OpenSSL PEPMA since its 58-bit numeric representation was not at the optimum level.

Future Work

Performance evaluation of PEPMA is a complex interdisciplinary research and thus, works involved with such multiple complexities will never be complete. One of the dimensions within interdisciplinary research is the uncertainties associated with their complexities. Before dealing more with such multidisciplinary exploration, it is necessary to acknowledge any missing or weaknesses of the findings in this study. Among the desirable findings listed in the result chapter, three essential findings as shown below in Table 64 have yet been fully realized. Those open deficiencies should be remedied to provide better accuracy in the evaluation. Realizing these additional measuring instruments suggests a variety of research to improve the combined key performance indicator; and the plan is to continue tackling these problems with future works and/or in the extension of this study making the performance evaluation more accurate.

Table 64. Future Work

Future Work (Why)	**Area of Enhancing**	**Rationale of Deficiency**
Evaluating Low-level Arithmetic and Arithmetic Optimization (To enhance efficiency measurement)	Including the findings and data analyses on six low-level mathematic routines as shown in Figure 30, block ⑨.	We could not provide such findings and data analyses due to limited scope of this paper
Enhancing Synchronization Agent (Improving accuracy for efficiency measurement)	Figure 21, BOCHS Hardware Emulation and Synchronization Agent	Virtual machine real-time response was slow
Evaluating Compliance Metric on every service module (To enhance compliance)	Compliance	Subjective and complex

Appendix A. Counting CPU Instructions

The cost indexes of *s_mpv_mul_d_add* () NSS PEPMA executable code:

Table 65. Cost Index of *s_mpv_mul_d_add*()

movq Count	movq Latency	movq Cost Index	imulq Count	imulq Latency	imulq Cost Index
29	6	174	4	10	40

Cost for functional computation is higher when cost index is higher.

The *s_mpv_mul_d_add* () NSS PEPMA 64-bit executable code compiled under GCC 4.7:

Table 66. The s_mpv_mul_d_add NSS PEPMA Executable Code

s_mpv_mul_d_add:			
.LFB122:			
	.cfi_startproc		
	pushq	%r12	#
	.cfi_def_cfa_offset 16		
	.cfi_offset 12, -16		
	pushq	%rbp	#
	.cfi_def_cfa_offset 24		
	.cfi_offset 6, -24		
	pushq	%rbx	#
	.cfi_def_cfa_offset 32		
	.cfi_offset 3, -32		
	movq	$0, carry(%rip)	#, carry
	testl	%esi, %esi	# a_len
	je	.L117	#,
	movq	%rdx, %r9	# b, D.6652
	shrq	$32, %r9	#, D.6652
	subl	$1, %esi	#, tmp90
	leaq	8(,%rsi,8), %r10	#, D.9447
	movl	$0, %eax	#, ivtmp.694
	andl	$4294967295, %edx	#, D.6649
	movabsq	$4294967296, %r11	#, tmp105
.L122:			
	movq	(%rdi,%rax), %r8	#* ivtmp.694, a_i.273
	movq	%r8, a_i(%rip)	# a_i.273, a_i
	movq	%r8, %rsi	# a_i.273, D.6648
	andl	$4294967295, %esi	#, D.6648
	movq	%r8, %rbp	# a_i.273, D.6651
	shrq	$32, %rbp	#, D.6651
	movq	%r9, %rbx	# D.6652, a0b1
	imulq	%rsi, %rbx	# D.6648, a0b1
	movq	%rdx, %r8	# D.6649, a1b0
	imulq	%rbp, %r8	# D.6651, a1b0
	addq	%rbx, %r8	# a0b1, a1b0.706
	movq	%r8, %r12	# a1b0.706, tmp92

	shrq	$32, %r12	#, tmp92
	imulq	%r9, %rbp	# D.6652, tmp93
	leaq	(%r12,%rbp), %rbp	#, a1b1.278
	movq	%rbp, a1b1(%rip)	# a1b1.278, a1b1
	cmpq	%r8, %rbx	# a1b0.706, a0b1
	jbe	.L118	#,
	addq	%r11, %rbp	# tmp105, tmp95
	movq	%rbp, a1b1(%rip)	# tmp95, a1b1
.L118:			
	salq	$32, %r8	#, a1b0.707
	imulq	%rdx, %rsi	# D.6649, a0b0.281
	leaq	(%rsi,%r8), %rsi	#, a0b0.281
	cmpq	%r8, %rsi	#a1b0.707, a0b0.281
	movq	a1b1(%rip), %rbx	# a1b1, tmp100
	adcq	$0, %rbx	#, tmp99
	movq	carry(%rip), %r8	# carry, carry.283
	addq	%r8, %rsi	#carry.283, a0b0.284
	movq	%rsi, a0b0(%rip)	# a0b0.284, a0b0
	cmpq	%r8, %rsi	#carry.283, a0b0.284
	adcq	$0, %rbx	#, tmp106
	movq	%rbx, a1b1(%rip)	# tmp106, a1b1
	movq	(%rcx,%rax), %r8	#* ivtmp.694, a_i.285
	movq	%r8, a_i(%rip)	# a_i.285, a_i
	addq	%r8, %rsi	# a_i.285, a0b0.286
	movq	%rsi, a0b0(%rip)	# a0b0.286, a0b0
	cmpq	%r8, %rsi	# a_i.285, a0b0.286
	adcq	$0, %rbx	#, tmp101
	movq	%rbx, a1b1(%rip)	# tmp101, a1b1
	movq	%rsi, (%rcx,%rax)	#a0b0.286,* ivtmp.694
	movq	a1b1(%rip), %rbx	# a1b1, a1b1
	movq	%rbx, carry(%rip)	# a1b1, carry
	addq	$8, %rax	#, ivtmp.694
	cmpq	%r10, %rax	# D.9447, ivtmp.694
	jne	.L122	#,
	addq	%r10, %rcx	# D.9447, c
.L117:			
	movq	carry(%rip), %rax	# carry, carry
	movq	%rax, (%rcx)	# carry,* c
	popq	%rbx	#
	.cfi_def_cfa_offset 24		
	popq	%rbp	#
	.cfi_def_cfa_offset 16		
	popq	%r12	#
	.cfi_def_cfa_offset 8		
	ret		

Appendix B. ECDH Protocol

The ECDH cryptography protocol used for exchanging private keys is believed to be intractable under an unsecured communication channel. Additionally, it is not feasible to find the discrete logarithm of a random 521-bit Elliptic curve element with respect to a publicly known base point G(*x*, *y*). The ECDH procedure starts out at transaction (1) with Client's domain parameters (*p*, *a*, *b*, G, *n*, *h*). The complete ECHD transaction under a public viewer and on an unsecured communication channel is summarized in the figure below. The scalar product calculation of *k*(*x*, *y*) occurs at the computations of *s*G, *c*G, *cs*G, where G(*x*, *y*) is the generator for the cyclic subgroup with order *n.* Furthermore, the scalar product *n*G(*x*, *y*) must equal to the infinity point O of the Elliptic curve; *h* is the cofactor that equals to the size of cyclic subgroup divided by *n*; *h* = E(*Fp*)/*n*

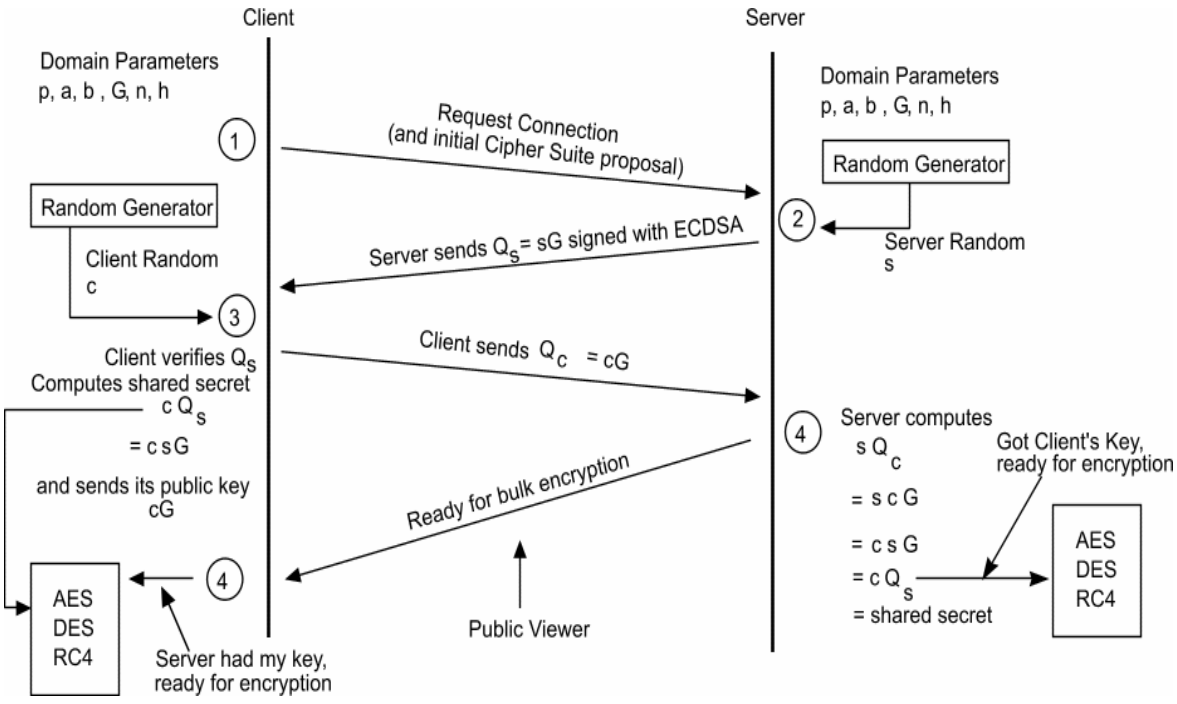

Figure 39. Elliptic Curve Diffie-Hellman Key Exchange Used with PEPMA

At time (1), Client initiates the session by sending a request signal followed by a proposed cipher and the client's PKE capability. Since both Server and Client have agreed on the cipher suite, they have the same domain parameters, including modulus[7] p; curve coefficient a and b; base point G(x, y); curve's order n; and cofactor h. At time (2), Server is ready to accept the connection and generates 521-bit random number s. Server then multiplies s with the base point G(x, y). This product is designated as $Q_s = s \times G(x, y)$.

The Server sends Q_s to the client (optionally with digital signature ECDSA). At time (3), Client receives Q_s and verifies that it has received the one sent from Server. Then Client computes the shared secret key: the product of two components $c \times Q_s$. Since Q_s equals to $s \times G(x, y)$, then the shared secret key $c \times Q_s$ must be $c \times s \times G(x, y)$.

The Client then proceeds to compute the public key $c \times G$ and sends it to the Server. This product is designated as $Q_c = c \times G(x, y)$. At time (4), Server receives Q_c and computes the product $s \times Q_c$. Equivalently, $s \times Q_c$ equals to $s \times c \times G(x, y)$; it also equals to $c \times Q_s$ – the shared secret of Client that sent from Client. At time (4), both parties have exchanged an elliptic curve based private key under the observation of public viewers.

On the Server side, PEPMA can help reduce computing costs when calculating sG and cQs. On the Client side, PEPMA can help reduce computing costs when calculating cG and cQs. Readers are referred to the next Appendix for a numerical example of this transaction.

---

[7] Modulus p is the same as modulus m in this research. It is a 521-bit prime.

Appendix C. An ECDH Transaction

The following data shows the results from an ECDH key negotiation corresponding to the transactions described in ECDH protocol.

```
base point x
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 C6
85 8E 06 B7 04 04 E9 CD 9E 3E CB 66 23 95 B4 42
9C 64 81 39 05 3F B5 21 F8 28 AF 60 6B 4D 3D BA
A1 4B 5E 77 EF E7 59 28 FE 1D C1 27 A2 FF A8 DE
33 48 B3 C1 85 6A 42 9B F9 7E 7E 31 C2 E5 BD 66

base point y
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 18
39 29 6A 78 9A 3B C0 04 5C 8A 5F B4 2C 7D 1B D9
98 F5 44 49 57 9B 44 68 17 AF BD 17 27 3E 66 2C
97 EE 72 99 5E F4 26 40 C5 50 B9 01 3F AD 07 61
35 3C 70 86 A2 72 C2 40 88 BE 94 76 9F D1 66 50

Server public key x
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 42
B5 EF EB 9C 79 89 77 5C F2 E4 B8 5F 0C EA 2E 2F
84 3D D7 DF 63 0E 9E 68 5F 9D 6B 0C F4 C7 9A A4
D9 83 7E C9 FB 53 B3 0D 3A 18 9E E3 50 4A 61 8D
47 55 FB 5A 88 C0 FF 3C 0F 73 A9 1D C5 AF 1D 60

Server public key y
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 27
62 AE 23 71 19 28 7A 17 DF 44 91 ED 14 F8 73 AD
4C BC 3F 6C C9 82 54 3B B5 07 CE 5D A4 AD E7 28
91 86 F3 D3 02 26 57 5E 70 54 A8 CC F5 E0 2B EF
D7 45 DA 26 CF 7C A9 8B A8 3B 4E DD 4D 25 2E 7D

Client private key x
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 21
FF 0E 29 45 9A 0B 2F 19 C0 81 C8 91 4E 30 8B 47
FF 8D 93 DD CC 06 BF 5D 20 70 82 73 55 7A 1F F1
73 44 F2 53 E7 1B 44 39 13 89 2C 60 43 7F 6F BD
15 D6 F2 8B EA 55 E1 30 CE 3D DC D9 A4 B9 F0 74
```

```
Client private key y
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 28
0A EA 55 12 25 26 44 1F 69 7C A9 F2 13 CF F3 3A
AB BF B6 25 BD C7 47 AC BA 2A 5E 20 5D BE E3 ED
9B D2 F5 0E C9 0B D7 F9 79 52 92 77 F8 94 88 8F
E8 BA 5C B7 2A 7D 95 55 28 6D C3 A9 8E 0D E9 E1

Client public key x
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 0F
D4 4B 13 C3 4A F1 9C DA E0 88 28 8D 5A 88 99 B1
67 23 6D 41 EE 77 1B 1D 06 64 AA 05 94 23 4A F1
78 A8 FB CA 5E 51 C0 AA 85 6C BB 3C E2 0C 10 B9
A1 ED 79 33 F0 0D BD 0A 2A 6B 87 F2 6F 06 43 84

Client public key y
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 DB
63 3D 86 7A 51 AB 8B D3 81 5F 50 B7 C5 5F 05 21
58 14 0A D0 D7 74 A4 1B 4B BC 91 C0 5A 5D 5C 86
D9 3C 54 34 4D 90 C8 EB 62 5A 28 98 76 00 6E 8C
7F D8 59 E9 19 B0 58 3B 4E A1 B6 D9 9F 87 FF 27

Server private key x, a.k.a. our key x
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 21
FF 0E 29 45 9A 0B 2F 19 C0 81 C8 91 4E 30 8B 47
FF 8D 93 DD CC 06 BF 5D 20 70 82 73 55 7A 1F F1
73 44 F2 53 E7 1B 44 39 13 89 2C 60 43 7F 6F BD
15 D6 F2 8B EA 55 E1 30 CE 3D DC D9 A4 B9 F0 74

Server private key y, a.k.a. our key y
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 28
0A EA 55 12 25 26 44 1F 69 7C A9 F2 13 CF F3 3A
AB BF B6 25 BD C7 47 AC BA 2A 5E 20 5D BE E3 ED
9B D2 F5 0E C9 0B D7 F9 79 52 92 77 F8 94 88 8F
E8 BA 5C B7 2A 7D 95 55 28 6D C3 A9 8E 0D E9 E1

Exchanged key x part was successful
Exchanged key y part was successful
```

The numeric example above shows the private key has two parts: *x* and *y*. The actual private key can be concatenated from *x* and *y*, making it a 1042-bit key.

Appendix D. Modulus m, Order m

In this research, italic letter "*m″* is used as a label for the Mersenne modulus of NIST P-521 Elliptic curve. Related literature might have used another letter to represent the modulus. One common designation from the industry is letter "*p*" for prime. Here, this letter "*p″* has already been designated as a Cartesian Elliptic-curve point $p(x, y)$.

Table 67. The Modulus of Finite Field

Format	$m = 2^{521} - 1$, $\log_2(m+1) = 521$
Hexadecimal	000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
Decimal	6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151
Number of Bits	521
Is *m* Prime?	Probably

The order of the cyclic subgroup is designated as italic letter "*n*".

Table 68. The Order of Finite Field

Hexadecimal	00001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409
Decimal	6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057746333217197532963996371363321113864768612440380340372808892707005449
Number of Bits	521
Is *n* Prime?	Probably

Appendix E. Point Adding of NSA Test Vectors

Input: *k, x, y*
Output: *k×(x, y)*

```
scalar k
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 EB
7F 81 78 5C 96 29 F1 36 A7 E8 F8 C6 74 95 71 09
73 55 54 11 1A 2A 86 6F A5 A1 66 69 94 19 BF A9
93 6C 78 B6 26 53 96 4D F0 D6 DA 94 0A 69 5C 72
94 D4 1B 2D 66 00 DE 6D FC F0 ED CF C8 9F DC B1

point x
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 D5
C6 93 F6 6C 08 ED 03 AD 0F 03 1F 93 74 43 45 8F
60 1F D0 98 D3 D0 22 7B 4B F6 28 73 AF 50 74 0B
0B B8 4A A1 57 FC 84 7B CF 8D C1 6A 8B 2B 8B FD
8E 2D 0A 7D 39 AF 04 B0 89 93 0E F6 DA D5 C1 B4

point y
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 44
B7 77 09 63 C6 3A 39 24 88 65 FF 36 B0 74 15 1E
AC 33 54 9B 22 4A F5 C8 66 4C 54 01 2B 81 8E D0
37 B2 B7 C1 A6 3A C8 9E BA A1 1E 07 DB 89 FC EE
5B 55 6E 49 76 4E E3 FA 66 EA 7A E6 1A C0 18 23

point x3
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 91
B1 5D 09 D0 CA 03 53 F8 F9 6B 93 CD B1 34 97 B0
A4 BB 58 2A E9 EB EF A3 5E EE 61 BF 7B 7D 04 1B
8E C3 4C 6C 00 C0 C0 67 1C 4A E0 63 31 8F B7 5B
E8 7A F4 FE 85 96 08 C9 5F 0A B4 77 4F 8C 95 BB

point y3
MSB.....................................LSB
|........| |........| |........| |........|
00 00 00 00 00 00 00 00 00 00 00 00 00 01 30
F8 F8 B5 E1 AB B4 DD 94 F6 BA AF 65 4A 2D 58 10
41 1E 77 B7 42 39 65 E0 C7 FD 79 EC 1A E5 63 C2
07 BD 25 5E E9 82 8E B7 A0 3F ED 56 52 40 D2 CC
80 DD D2 CE CB B2 EB 50 F0 95 1F 75 AD 87 97 7F
```

Appendix F. NIST Test Vectors

```
#  CAVS 11.0
#  "Key Pair" information
#  Curves selected: P-192 P-224 P-256 P-384 P-521 K-163 K-233 K-283 K-409 K-571
B-163 B-233 B-283 B-409 B-571
#  Generated on Wed Mar 16 16:16:42 2011
[P-521]
[B.4.2 Key Pair Generation by Testing Candidates]
N = 10
d =
0184258EA667AB99D09D4363B3F51384FC0ACD2F3B66258EF31203ED30363FCDA7661B6A817DA
AF831415A1F21CB1CDA3A74CC1865F2EF40F683C14174EA72803CFF
Qx =
019EE818048F86ADA6DB866B7E49A9B535750C3673CB61BBFE5585C2DF263860FE4D8AA8F7486
AED5EA2A4D733E346EAEFA87AC515C78B9A986EE861584926CE4860
Qy =
01B6809C89C0AA7FB057A32ACBB9AB4D7B06BA39DBA8833B9B54424ADD2956E95FE48B7FBF6
0C3DF5172BF386F2505F1E1BB2893DA3B96D4F5AE78F2544881A238F7

d =
014B967F6651B5E6A482FCCC609AB6630B3806FE1F94F4083319B0B50575FB3436A04F508172F7F
C396D6E969CA3E8D1C1E9A84D431A48B94F30566DC6808DD1D138
Qx =
0145F371040D3D4A24D6D3CEB2681DB207B77096AB57606D92981A69CE35A0AC4628C2DC1284E
4DD9715CDE46F18B59E9FC98FEA162CEB6E2C481ECBFAD4E19D3ABF
Qy =
0125EB751FF4FB8BB98E1FB455D2CFB35E3323DE5C7280FC9E51729704F4FEC51D5A6CE6C1F75
DBF710E1F9D3EE9F2A77E7C12C045E729D0E9A281C37F0F07B8CF0C

d =
7616133442038E27357DB450C353BD11FBA3BCAC8B7B8C3EF76AADB5FE05BE1DD57A22D42A5
444D00DCD018D389170C54FE781CB21C36020F657D001E1CBB41DD1
Qx =
BBECF65446053080CC1CF955938C58EB630C84ECAD2756F93B47EBFA9F9BCA3FA834353981260
8CAB2D3A9F8079AB8311A4F269B0A3CD9E0DDD066FC4121D92F0E
Qy =
01DD96DB411AD67997B10D42C76B8510C8A930DFA9A5927AC274B0C5021798690777B8E77E6AE
2648BF513E02F586898E7DAE20D71D19838A9F3175F06B057C5F2F4

d =
013BCC0ED286861D3F5463BCFC0B68A6EC0FCF86291BA41257838B72536ADA986E43E05EC4C32
C0B29DA632DD1CE39EFC81C8278F5D18D9CF27F6E75523821A46D99
Qx =
A3A165C2BB535D1041D54B749E2F6E6C734A03C09DF69C14A5DD2AA57790ACC504548885F0BD
3A44F8B66BB9C36B3FF257D7D465EFB81445D4CC5A5AF7F36C679C
Qy =
8A5D094E4F2AA18FB877D2649DFD76F9482AC2E049AEFBB463F3C9061CFDFAEC785DF9577A09
0E45A17330F422FB16A16ACCCFF9ADE7B034EC544C7A8AEA441C49

d =
01F79977450CE5887AE2EF7D648AB658C056E57F0A690CF28A4E94F373F2C15EB3C0D3E0D670FE
CA6FF02D5FD03187146EB85E09D72F8CABB1900D0C338A23080C12
```

Qx =
016D9EDE24A3950098798766E57C53F2749CD0D3F56CA0904A3711C030965291EDD5C6FE0903771
768F42340E88E1CD2F161358972775FA53E5B87C3B660ACC447E2
Qy =
010CA5CFE6DF8E069AE1326DD9E18CCA75CDE7CB24B427A409025F9E12B5098A56A20BB90B1
D23B75FAD7A54F9E25FF892E1236D1717F1F94E18FA2289F899FE2221

d =
9A9160D2614937C33284627826BE871C26407C84D23E6D23DE5F5F48B500B89B0BC07F10C4E0FB9
9C085D9E9D7149278F76E3FAE4ABAEEEF2495FE3D228EF0F949
Qx =
019ED72E6BFC673F2A852ACF9D60E2C3B19C50A56C54AC304612B26F83AFE1AFF4F87DCA458E
83B6F89EC48F8B1A20931ACD3C97C71BF21B5633CF4FD68437DB45C1
Qy =
D141DC4272CA03A528AD8FDADE9ECB3070FB2D4AF0BB296ABDAED651B5D26573EB4443A4D
0D4134FF248D8ED402C93BF6A905CB2792B9CECB4AEB69ED78F410382

d =
2FDC02492573228ADA3FA8A2DB68D72E9396A2BFCA9A8EBDB5C2955CC894A7493CFAE001759
368EB8FFC3C29B15365F6484CDD6A44E084F1D3C88DBA7AA4F29C3C
Qx =
010BA48733FC3E8F54F601F74659BCD43FDE4CF8C5A07DA341CE68E792F8F70721C23DC6D9B1B
401BD3254C8DE546E9367F10AEE947B1DD295E6D822524546DDC195
Qy =
01B2C0EA5C4171CDC069FC6C69E18636CFA404F487A143B3981A1F212969CDBD6601A84302867F
8A4A4730FDCD0F994C226F7C02C5E664B79C34B7E5D071423FF528

d =
01AD69406C11C66FAD5FE2295F0E526622488755ECB18BA12EE51FA879ED47FF5F5B05195A821E
8D36489492B5DE2009F303E17B9FDF6379DAE52C0178A16927CA38
Qx =
01F1CA24041BA73812C1124E96454545C45AB903407AFCE3105108362ED3CB4F7D0D5B1466074C
2EF22C7FD1EBC16E74A74A163FBB2F530EF44549DAD81E806F24D6
Qy =
6B34D6EFF12BB76AEE9BD7AC590E437735AE77DA4A60191E8E01F1CEB8AD7C1EDA4D0F84D4
ED2DC72DE702D351EF8F64B2CDF2A95EF185D3119F276F6CCB3C5A65

d =
013C41B6514C608A2E4696CFC6BD2DDD36611CA5DBF6F2D2E3E32A1925C5AE4FF591DCAA75C
4E8043ADCB99D510CB664868BB638A2C52B81BB240A974548A68FCE79
Qx =
C6D82F16433C71E37F2E9779BE4599A3B1DDA415F6C338E52DF4CA70607A69637B50170F21BBB7
F60B9A9C145BB63E6D4F370FCD00BFB60F7A0DC55CC44F65FC90
Qy =
0152344D6F2E72DEB2C59FF2AE268FB067279A1942AE231734BA980C5457A6A73BBF2B13343AE4
4A0C8A712572851DA4B91065EE0436ABE811AE71883C4A2F1B797F

d =
316E2D06FD00C9C4266EA20BF60CDF867859A6F5BA242DE35054CDCF5486E5E344AB1D1BCE13
E2CC831137320774EC3AB0F6FB554FCCEC56ADA267959794898028
Qx =
A183880E61C6E0435E591694E51F63C099FCD5B61E3DDACC4057399AFC6A90321424AB0EC1699
AEEB9C404616D62C23466132B52583C18D3530116B58AD41452F0
Qy =
191E06057E2282B4DE6E0741FB37B04F0E6AE172BE81267B0DB3023E7A116AC5861DECD54BA84
E15D5FD64D6CA628461B79E120851BED1C74ADEBE3DDEE838A170

Appendix G. NSS Exponentiation Procedure

Users will find NSS PEPMA in its source code repository (NSS-1, 2014). The NSS exponentiation procedure is coded in the source file "ecp_jac.c". NSS PEPMA makes 524 calls to the point-doubling and 131 calls to the point-adding function. The following messages list some essential entrant parameters to the PEPMA:

```
GFMethod_consGFp
modulus =
1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFF
order =
1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFA51868783BF2F966B7FCC0148F709A5D03BB5C9B8899C47AEBB6FB71E91386
409
a =
1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFC
b =
51953EB9618E1C9A1F929A21A0B68540EEA2DA725B99B315F3B8B489918EF109E
156193951EC7E937B1652C0BD3BB1BF073573DF883D2C34F1EF451FD46B503F00
genx =
81D687818FC9BA21A2C7B00FE84C69E8113DA5FB1439D7A83A0585DC2550ABF
E423DE7C6E0B54595C5FEC716853E9CD7825844C9877B6D6AE0DA7571A4FD9B0
23B
geny =
144B7770963C63A39248865FF36B074151EAC33549B224AF5C8664C54012B818ED
037B2B7C1A63AC89EBAA11E07DB89FCEE5B556E49764EE3FA66EA7AE61AC01
823
scalar d =
1EB7F81785C9629F136A7E8F8C67495710973555411 1A2A866FA5A166699419BFA
9936C78B62653964DF0D6DA940A695C7294D41B2D6600DE6DFCF0EDCFC89FDC
B1
ec_GFp_pt_mul_jac
mp_digit size = 8
unsigned int = 4
unsigned long = 8
unsigned long long = 8
Exiting ec_GFp_pt_mul_jac
```

Total double = 524, total add = 131
Total double bypass = 0, total add bypass = 1

result x =
4634C62EEC4F2D875DE95AE71E95C58812342A29A37139A23299F053203F9BE9D
00E057C6CBBFB8D6C6D9EFDBE34BA409949AD09809B15A98A08637136CE7239
37
result y =
162495DB3E31BE7E8EEFCD96EE6698EB915DE882118DFAD1BC83B1369FAB93A
35D69C0E9A7A3CA1D7F83ED2EE2FBFD565AAF76A65BBC1C1C7C97CFF45DD7
B533FAF

An incomplete listing of NSS PEPMA is provided below for reference.

NSS PEPMA:

```
mp_err ec_GFp_pt_mul_jac(const mp_int *n, const mp_int *px, const mp_int *py,
                         mp_int *rx, mp_int *ry, const ECGroup *group)
{
    mp_err res = MP_OKAY;
    mp_int precomp[16][2], rz;
    int i, ni, d;

    MP_DIGITS(&rz) = 0;
    for (i = 0; i < 16; i++) {
      MP_DIGITS(&precomp[i][0]) = 0;
      MP_DIGITS(&precomp[i][1]) = 0;
    }

    ARGCHK(group != NULL, MP_BADARG);
    ARGCHK((n != NULL) && (px != NULL) && (py != NULL), MP_BADARG);

    /* initialize precomputation table */
    for (i = 0; i < 16; i++) {
      MP_CHECKOK(mp_init(&precomp[i][0]));
      MP_CHECKOK(mp_init(&precomp[i][1]));
    }

    /* fill precomputation table */
    mp_zero(&precomp[0][0]);
    mp_zero(&precomp[0][1]);
    MP_CHECKOK(mp_copy(px, &precomp[1][0]));
    MP_CHECKOK(mp_copy(py, &precomp[1][1]));
```

```
for (i = 2; i < 16; i++) {
        MP_CHECKOK(group->
                        point_add(&precomp[1][0], &precomp[1][1],
                                                &precomp[i - 1][0], &precomp[i - 1][1],
                                                &precomp[i][0], &precomp[i][1], group));
    }

    d = (mpl_significant_bits(n) + 3) / 4;

    /* R = inf */
    MP_CHECKOK(mp_init(&rz));
    MP_CHECKOK(ec_GFp_pt_set_inf_jac(rx, ry, &rz));

    for (i = d - 1; i >= 0; i--) {
      /* compute window ni */
      ni = MP_GET_BIT(n, 4 * i + 3);
      ni <<= 1;
      ni |= MP_GET_BIT(n, 4 * i + 2);
      ni <<= 1;
      ni |= MP_GET_BIT(n, 4 * i + 1);
      ni <<= 1;
      ni |= MP_GET_BIT(n, 4 * i);
      /* R = 2^4 * R */
      MP_CHECKOK(ec_GFp_pt_dbl_jac(rx, ry, &rz, rx, ry, &rz, group));
      MP_CHECKOK(ec_GFp_pt_dbl_jac(rx, ry, &rz, rx, ry, &rz, group));
      MP_CHECKOK(ec_GFp_pt_dbl_jac(rx, ry, &rz, rx, ry, &rz, group));
      MP_CHECKOK(ec_GFp_pt_dbl_jac(rx, ry, &rz, rx, ry, &rz, group));
      /* R = R + (ni * P) */
      MP_CHECKOK(ec_GFp_pt_add_jac_aff
                        (rx, ry, &rz, &precomp[ni][0], &precomp[ni][1], rx, ry,
                              &rz, group));
    }

    /* convert result S to affine coordinates */
    MP_CHECKOK(ec_GFp_pt_jac2aff(rx, ry, &rz, rx, ry, group));

  CLEANUP:
    mp_clear(&rz);
    for (i = 0; i < 16; i++) {
      mp_clear(&precomp[i][0]);
      mp_clear(&precomp[i][1]);
    }
    return res;
}
```

Appendix H. OpenSSL Exponentiation Procedure

Users will find OpenSSL PEPMA in its source code repository (OpenSSL-1, 2014). Source code containing Elliptic-curve service routines is included only in compressed file (tar) with Elliptic-curve capability. The OpenSSL PEPMA exponentiation procedure is coded in the source file "ecp_nistp521.c". By executing the code below, OpenSSL PEPMA makes 520 calls to the point-doubling and 104 calls to the point-adding function. An incomplete listing of OpenSSL PEPMA is provided for reference as follows:

```
static void batch_mul(felem x_out, felem y_out, felem z_out,
        const felem_bytearray scalars[], const unsigned num_points, const u8 *g_scalar,
        const int mixed, const felem pre_comp[][17][3], const felem g_pre_comp[16][3])
        {...
for (i = (num_points ? 520 : 130); i >= 0; --i)
          {
          /* double */
          if (!skip)
                  point_double(nq[0], nq[1], nq[2], nq[0], nq[1], nq[2]);

          /* add multiples of the generator */
          if (gen_mul && (i <= 130))
                  {
                  bits = get_bit(g_scalar, i + 390) << 3;
                  if (i < 130)
                          {
                          bits |= get_bit(g_scalar, i + 260) << 2;
                          bits |= get_bit(g_scalar, i + 130) << 1;
                          bits |= get_bit(g_scalar, i);
                          }
                  /* select the point to add, in constant time */
                  select_point(bits, 16, g_pre_comp, tmp);
                  if (!skip)
                          {
                          point_add(nq[0], nq[1], nq[2],
                                  nq[0], nq[1], nq[2],
                                  1 /* mixed */, tmp[0], tmp[1], tmp[2]);
                          }
                  else
                          {
                          memcpy(nq, tmp, 3 * sizeof(felem));
                          skip = 0;
```

```
                }
        }

    /* do other additions every 5 doublings */
    if (num_points && (i % 5 == 0))
        {
        /* loop over all scalars */
        for (num = 0; num < num_points; ++num)
            {
            bits = get_bit(scalars[num], i + 4) << 5;
            bits |= get_bit(scalars[num], i + 3) << 4;
            bits |= get_bit(scalars[num], i + 2) << 3;
            bits |= get_bit(scalars[num], i + 1) << 2;
            bits |= get_bit(scalars[num], i) << 1;
            bits |= get_bit(scalars[num], i - 1);
            ec_GFp_nistp_recode_scalar_bits(&sign, &digit, bits);

            /* select the point to add or subtract, in constant time */
            select_point(digit, 17, pre_comp[num], tmp);
            felem_neg(tmp[3], tmp[1]); /* (X, -Y, Z) is the negative point */
            copy_conditional(tmp[1], tmp[3], (-(limb) sign));

            if (!skip)
                {
                point_add(nq[0], nq[1], nq[2],
                        nq[0], nq[1], nq[2],
                        mixed, tmp[0], tmp[1], tmp[2]);
                }
            else
                {
                memcpy(nq, tmp, 3 * sizeof(felem));
                skip = 0;
                }
            }
        }
    }
felem_assign(x_out, nq[0]);
felem_assign(y_out, nq[1]);
felem_assign(z_out, nq[2]);
}
```

Appendix I. Description of *Clock()* Function

The *clock()* function provides an elapsed CPU time used by a running process. In measuring the elapsed processing time, PEPMA calls the clock function *clock()* at the beginning and at the ending of the executing interval. PEPMA subtracts the end_time from start_time to obtain the absolute elapsed_time. It then divides the absolute time by CLOCKS_PER_SEC. A typical setup in Linux environment is shown below:

```
#include <time.h>

clock_t start_time, end_time;
double elapsed_time;

    start_time = clock();
    for (loop_count...;)
      {
            // do EPM (calling point-adding, point-doubling functions etc.)
      }
    end_time = clock();
    elapsed_time =
    ((double) (end_time - start_time))/(CLOCKS_PER_SEC * loop_count);
```

The constant CLOCKS_PER_SEC defines the number of ticks per second. In a Linux system, CLOCKS_PER_SEC is an integer value normally equates to 1000. The data type of "clock_t" is equivalent to "long int" which is 64-bit integer in a 64-bit x 86 computing platform.

The purpose of using "for loop" with loop count is to increase measurement precision.

Appendix J. Selection of Operational Parameters for P-521

The listing below is an incomplete set of Request-For-Comment (RFC) introducing Elliptic-curve Cryptography into the cyber-space security system. In the contexts of these RFCs, the selection of P-521 was analyzed and proposed for which operational parameters of curve P-521 would be best suited to use or to exclude. Besides the security requirements, some of the important recommendations for selecting the operational parameters of P-521 curve were the efficiency and ease of implementation of underlying arithmetic.

Table 69. Request-For-Comment Related to Selection of P-521 Curve

Request-For-Comment	**Year**	**Discussion of**
RFC-6637: Elliptic Curve Cryptography (ECC) in OpenPGP	2012	NIST ECC curve P-521 Profile
RFC-5639: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation	2010	Operational parameters of curve P-521
RFC-3766: Determining Strengths For Public Keys Used For Exchanging Symmetric Keys	2004	Choosing parameters for the equation

OpenPGP: Email encryption standard, open-source Pretty-Good-Privacy

As stated in RFCs listed above, the AES-256 symmetric key encryption system requires an asymmetric Elliptic-curve key length around 512 to 576 bits. Since there is only one unique Mersenne probable prime of length 521 bits, all standards should converge to the arithmetic using 521 bits. It has been known that the efficiency and easiness of modulo arithmetic can be obtained with a Mersenne prime (Hankerson et al., 2004, pp. 44-46).

The American National Standards Institute (ANSI), National Security Agency (NSA), National Institute of Standards and Technology (NIST), and Standards for Efficient Cryptography Group (SECG) authorities published their own selection of curves and underlying arithmetic. Within the context of security and computing efficiency, they recommended to the industry what and how to apply operational parameters. The

publication listing below is an incomplete set of standards recommending the implementation of underlying arithmetic for P-521 curve.

Table 70. ANSI, NSA, NIST, and SECS Publications

Standards	**Year**	**Discussion of**
FIPS PUB 186-4: Digital Signature Standard (FIPS PUB 186-4, 2013). NIST	Jul/ 2013	P-521 curve, efficiency, arithmetic approach, and modulo reduction, and projective transformation in depth
NSA Suite B (NSA, 2013)	2013	P-521 curve, efficiency, and arithmetic approach
SECG: Standards for Efficient Cryptography Group (SEC 1, 2000)	Jan/ 2010	P-521 curve, efficiency, arithmetic approach, modulo reduction
ANSI X9.62 (ANSI, 2005)	2005	General Elliptic-curves and arithmetic

The 521-bit Mersenne prime has a unique property that can be written as the sum or difference of a small number of powers of 2. For example, a 521-bit Mersenne prime has an integer value of $p = 2^{521} - 1$. This unique property offers a fast reduction algorithm on computing platforms with machine word size = 64 bits; the arithmetic for modulo reduction requires only additions and subtractions (Hankerson et al., 2004, pp. 44-46).

Appendix K. Operation of BOCHS

BOCHS is a piece of software to emulate a virtual machine. Virtualization allows code and data of PEPMA to execute within a newly created and isolated runtime environment. In this study, BOCHS runs under CENTOS 6.4, a Linux variation Operating System (OS). Because this CENTOS 6.4 OS runs at the lowest level of a hardware platform, thus, it is known as the host Operating System. In this study, BOCHS emulates CENTOS 6.0. Then this OS becomes a guest OS, which provides all necessary operating system resources to execute PEPMA in a rescue mode. This rescue operating mode provides a minimum but adequate set of peripheral and working environment. To start BOCHS, execute the bash script as follows:

/BOCHS/bochs -q -f c6.txt

/BOCH/ is a directory where BOCHS installed and “c6.txt” is the configuration file for “bochs” program. When BOCHS starts successfully, a welcome screen will appear as follows:

Figure 40. Virtual Machine BOCHS Main Screen

Press <ESC> to bring in next screen, and enter “linux rescue” as shown on the screen below (without entering double quotes).

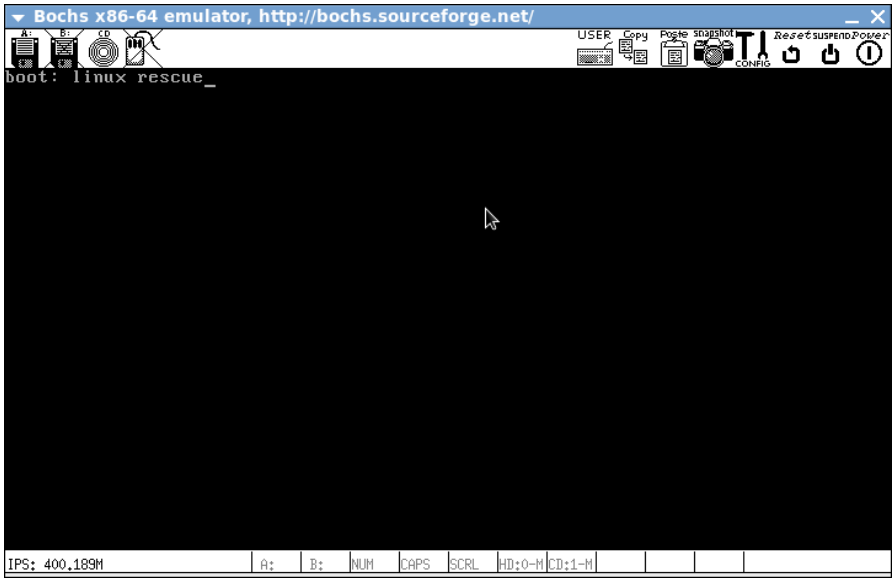

Figure 41. Virtual Machine BOCHS Rescue Screen

When host OS and BOCHS bring in next screen, the emulation has been going successfully up to this point:

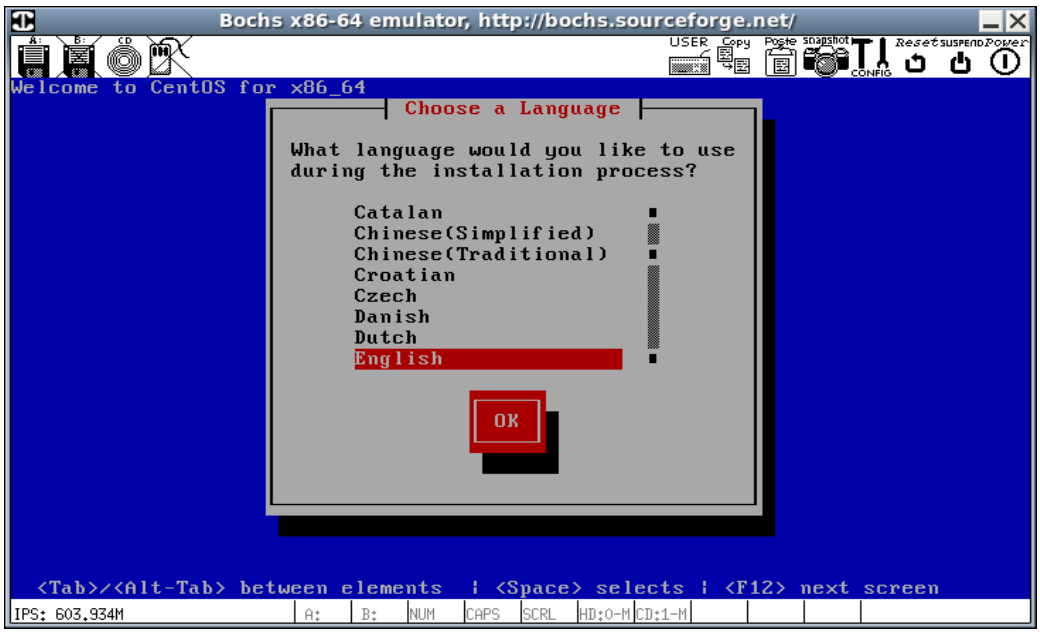

Figure 42. Virtual Machine BOCHS Language Screen

When BOCHS asks for enabling the network, enter “no”, enter “skip” for checking rescue environment, then enter “shell start” at the menu to start bash shell. After this point, BOCHS loads the terminal and ready for commands. If this terminal screen shows up with “bash-4.1#” prompt, the virtualization has been completely successful.

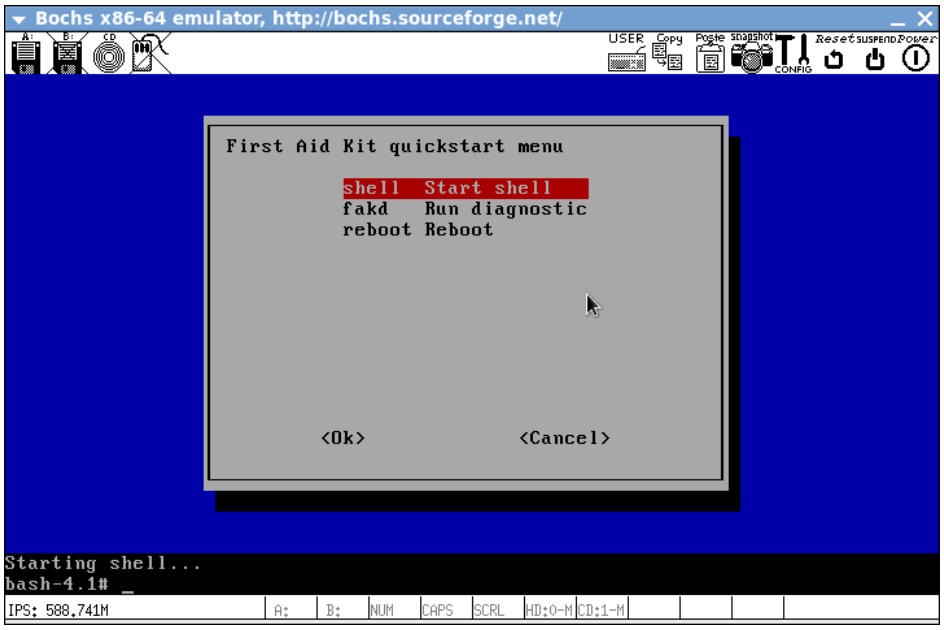

Figure 43. Virtual Machine BOCHS Final Screen

To execute PEPMA, enter the commands at bash prompt as follows:

cd /mnt

mkdir f

mount –t ext2 /dev/fd0 /mnt/f

/mnt/f/ectest

/mnt/f/ecp_test

A typical display from machine emulation is shown in the picture below. This screen shows the "bash" terminal in virtual machine BOCHS. The texts shown on the screen are the results from PEPMA exponentiation function calculating the scalar product $k\times(x, y)$. Counting of CPU instructions (MULq, MOVq etc.) are outputted on the host-machine terminal.

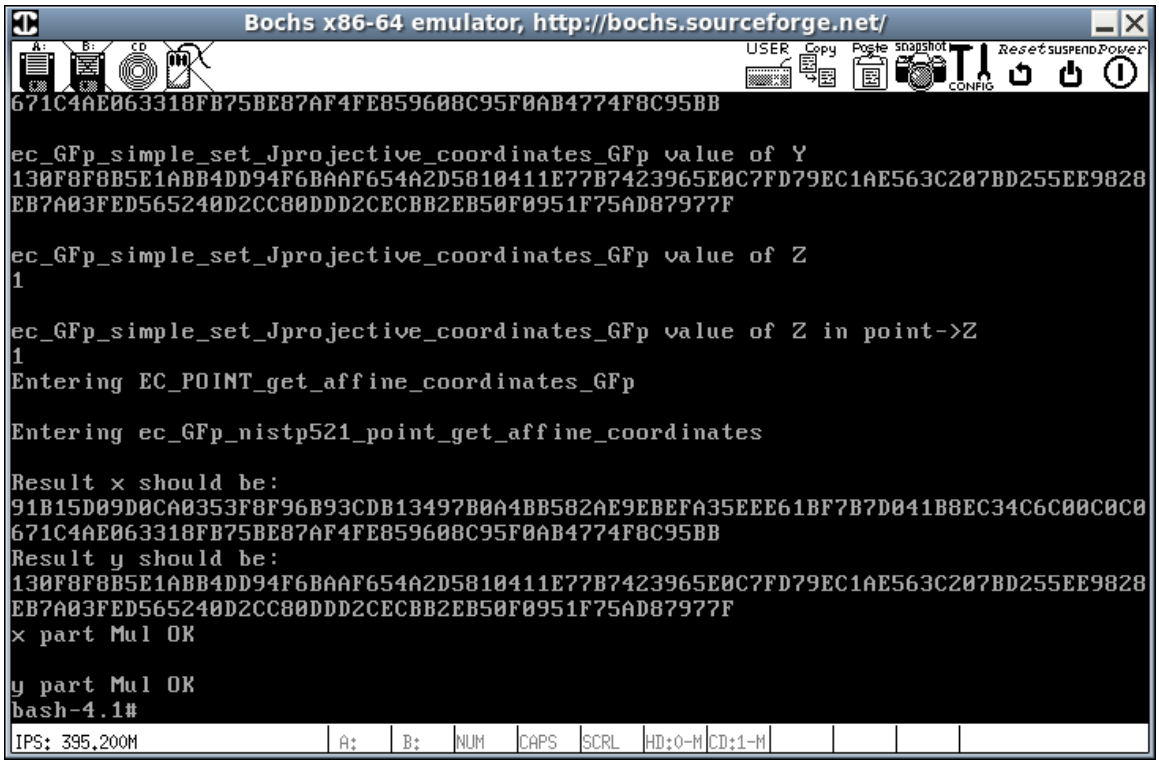

Figure 44. Virtual Machine BOCHS Calculating $k\times(x, y)$

From the figure shown below, the terminal running on host OS displays selected software-counters of unit-under-test. Target identification (0001BF75) is located at the first parameter; and its executing thread is located at the second parameter (00006400). BOCHS counts the number of MULq and MOVq CPU instructions and displays them at the third parameter (00001E6D) and the last parameter (0005A362) respectively.

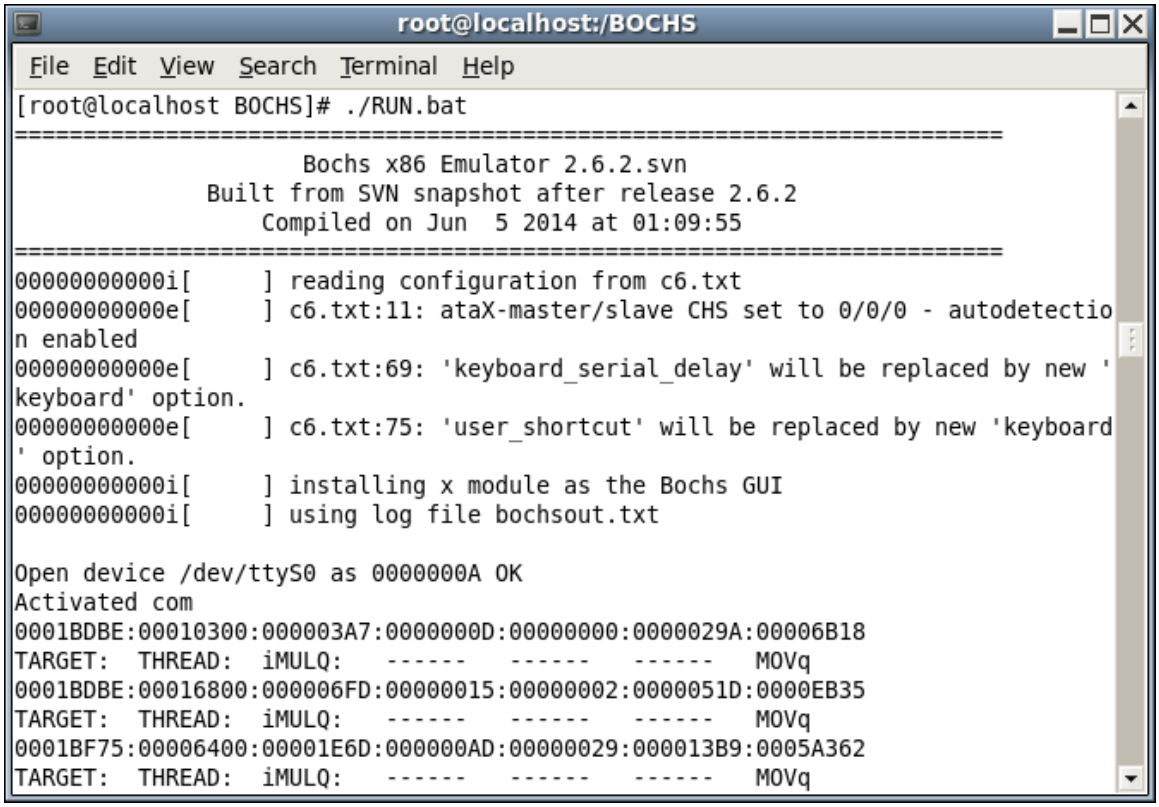

Figure 45. Instruction Software Counters Displayed while Calculating $k\times(x, y)$

Update Image

To update PEPMA image from host OS, enter the commands at guest OS bash prompt as follows:

umount /mnt/f

Enter the commands at host OS bash prompt as follows:

```
cd /BOCHS
losetup /dev/loop0 a.img
mount -t ext2 /dev/loop0 -o loop /mnt/floppy
```

Then copy OpenSSL PEPMA (ectest) or NSS PEPMA (ecp_test) to the guest Disk (executing commands from host OS terminal):

```
cp /O/test/ectest /mnt/floppy/ectest
cp /NSS/mozilla/security/nss/lib/freebl/ecl/ecp_test /mnt/floppy/ecp_test
umount /dev/loop0
losetup -d /dev/loop0
```

Content of c6.txt

```
megs: 512
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
floppya: 1_44=a.img, status=inserted
floppyb: 1_44=b.img, status=inserted
ata0-master: type=disk, path="c6min.img", mode=flat
ata1-master: type=cdrom, path=./c6min.iso, status=inserted
boot: cdrom
# this simulates  /dev/ttyS0 on guest
# com1: enabled=1, mode=file, dev=serial.out
# this simulates  /dev/ttyS0 on guest
# com1: enabled=1, mode=term, dev=/dev/pts/0
# com1: enabled=1, mode=term, dev=/dev/tty0
com1: enabled=1, mode=term, dev=/dev/ttyS0
# panic: action=ask
panic: action=report
error: action=report
info: action=report
debug: action=ignore
# ne2k: ioaddr=0x300, irq=9, mac=00:c4:3B:00:C3:00, ethmod=win32, ethdev=NE2000
# default config interface is textconfig.
```

```
#config_interface: textconfig
#config_interface: wx
display_library: x
# other choices: win32 sdl wx carbon amigaos beos macintosh nogui rfb term svga
log: bochsout.txt
mouse: enabled=0, type=ps2
fullscreen: enabled=0

cpu: ips=400000000, ignore_bad_msrs=1
clock: sync=both
keyboard_serial_delay: 250
keyboard: keymap=$BXSHARE/keymaps/x11-pc-us.map
# keyboard_paste_delay: 100000
user_shortcut: keys="f7"
# mouse: enabled=1
#magic_break: enabled=1
#port_e9_hack: enabled=1
#text_snapshot_check: enabled=0
#private_colormap: enabled=0
```

BOCHS Configuration and Compilation

```
#  ./configure --enable-cpu-level=6 \
#              --enable-smp \
#              --enable-x86-64 \
#              --enable-pci \
#              --enable-disasm \
#              --enable-logging \
#              --enable-cdrom \
#              --disable-plugins \
#              --enable-usb \
#              --enable-usb-ohci \
#              --enable-usb-xhci \
#              --enable-plugins \
#              --enable-vmx \
#              --enable-fpu \
#              --enable-debugger \
#             --with-x --with-x11 --with-term
#
# --disable-assert-checks \
#            --enable-debugger \
#              --enable-debugger-gui \
#              --enable-sb16 \
#
#./configure --enable-cpu-level=6 \
#--enable-ne2000 \
```

```
#--enable-pci \
#--enable-pcidev \
#--enable-pnic \
#--enable-repeat-speedups \
#--enable-fast-function-calls \
#--enable-all-optimizations \
#--enable-fpu \
#--enable-cdrom \
#--enable-x86-64 \
#--with-x --with-x11 --with-term
./configure --enable-cpu-level=6 \
--disable-smp \
--enable-ne2000 \
--enable-pci \
--enable-pcidev \
--enable-pnic \
--enable-repeat-speedups \
--enable-cdrom \
--enable-x86-64 \
--with-x --with-x11 --with-term
#./configure --enable-cpu-level=6 \
#--enable-x86-64 \
#--enable-pci \
#--enable-pcidev \
#--enable-debugger \
# --with-x --with-x11 --with-term

# if this error happens while compiling:
#gui/libgui.a(gtk_enh_dbg_osdep.o): In function `MakeGTKthreads()':
#./build/bochs-2.4.2/gui/gtk_enh_dbg_osdep.cc:2120:
# undefined reference to `pthread_create'
# add this statement -lpthread to Makefile under LIBS (around line 100)
#
# Put text above into bash file RUN.bat
# Compile BOCHS by executing ./RUN.bat
# alias m = "make"
# alias r = "./RUN.bat"
```

Appendix L. Operation of PAPI

The Performance Application Programming Interface, PAPI, is a machine independent set of callable routines that provide access to the hardware performance counters inside a CPU. It is currently being developed at the University of Tennessee, and the codes are mostly written in C language. To use PAPI counting services, PEPMA links to code library "libpapi.a". Compilation of PEPMA must include a PAPI header, papi.h, declared variables, and the library path to the "libpapi.a":

```
#include "/usr/local/include/papi.h"

#define NUM_EVENTS 2
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: \n",
retval,__FILE__,__LINE__);  exit(retval); }
int EventSet = PAPI_NULL;
long long papi_values[NUM_EVENTS];
char errstring[PAPI_MAX_STR_LEN];
int retval;
```

Then before measuring, initialize PAPI and create counting events with:

```
if((retval = PAPI_library_init(PAPI_VER_CURRENT)) != PAPI_VER_CURRENT )
ERROR_RETURN(retval);
  /* Creating the EventSet */
  if ( (retval = PAPI_create_eventset(&EventSet)) != PAPI_OK)
    ERROR_RETURN(retval);


  /* Add Total Instructions executed to the EventSet */
  if ( (retval = PAPI_add_event(EventSet, PAPI_TOT_INS)) != PAPI_OK)
    ERROR_RETURN(retval);


  /* Add Total Cycles executed to the EventSet */
  if ( (retval = PAPI_add_event(EventSet, PAPI_TOT_CYC)) != PAPI_OK)
    ERROR_RETURN(retval);
```

Start the PAPI measuring instrument by calling function PAPI_start() just before executing the unit-under-test Exponentiation Function (EF):

```
if ( (retval = PAPI_start(EventSet)) != PAPI_OK) ERROR_RETURN(retval);
      EF()...
```

Then read the PAPI instrument with PAPI_read() to acquire the results:

```
  /* Read the counter values and store them in the values array */
  if ( (retval=PAPI_read(EventSet, papi_values)) != PAPI_OK)
    ERROR_RETURN(retval);
  /* Stop counting and store the values into the array */
  if ( (retval = PAPI_stop(EventSet, papi_values)) != PAPI_OK)
    ERROR_RETURN(retval);
  printf("\nTotal instructions executed are %lld", papi_values[0] );
  printf("\nTotal cycles executed are %lld \n",papi_values[1]);
  /* Free the resources used by PAPI */
  PAPI_shutdown();
```

Appendix M. Configuration and Compilation of NSS

Execute the following commands to compile NSS:

```
cd /NSS/mozilla/security/nss
# printenv $CFLAGS
# read -p "Press any key to continue"
unset NSS_ENABLE_ECC
unset NSS_ECC_MORE_THAN_SUITE_B
unset ECL_ENABLE_GFP_PT_MUL_JAC
unset BUILD_OPT
unset NSS_USE_COMBA
NSS_USE_COMBA=0
export NSS_USE_COMBA
# NSS_ENABLE_ECC=1
# export NSS_ENABLE_ECC
NSS_ECC_MORE_THAN_SUITE_B=1
export NSS_ECC_MORE_THAN_SUITE_B
USE_64=1
export USE_64
ECL_ENABLE_GFP_PT_MUL_JAC=1
export ECL_ENABLE_GFP_PT_MUL_JAC
# no debug
#BUILD_OPT=1
#export BUILD_OPT
# this flag does not work well yet
#
unset ECL_ENABLE_GFP_PT_MUL_JAC
make clean
NSS_ECC_MORE_THAN_SUITE_B=1
# make nss_build_all USE_64=1 NSS_ENABLE_ECC=1
make nss_build_all USE_64=1 NSS_ECC_MORE_THAN_SUITE_B=1
ECL_ENABLE_GFP_PT_MUL_JAC=1

cd /NSS/mozilla/security/nss/lib/freebl/ecl
alias n="make clean"
alias m="make tests"
alias r="./ecp_test --print --time"
```

Appendix N. Configuration and Compilation of OpenSSL

Execute the following commands to configure OpenSSL for compilation:

```
./config enable-ec_nistp_64_gcc_128
#make depend
#make

# add this line in make file
# for PAPI
# EX_LIBS= /usr/local/lib64/libpapi.a

cd /O
alias m="make"
alias r="./test/ectest"
```

Appendix O. Test Vector Type A

scalar *k* =
1EB7F81785C9629F136A7E8F8C67495710973555411 1A2A866FA5A166699419BFA9936C78B62653964DF0D6DA940A695C7294D41B2D6600DE6DFCF0EDCFC89FDCB1

affine coordinate *x* =
1D5C693F66C08ED03AD0F031F937443458F601FD098D3D0227B4BF62873AF50740B0BB84AA157FC847BCF8DC16A8B2B8BFD8E2D0A7D39AF04B089930EF6DAD5C1B4

affine coordinate *y* =
144B7770963C63A39248865FF36B074151EAC33549B224AF5C8664C54012B818ED037B2B7C1A63AC89EBAA11E07DB89FCEE5B556E49764EE3FA66EA7AE61AC01823

The results of function $k(x, y)$ are:

result *x* =
91B15D09D0CA0353F8F96B93CDB13497B0A4BB582AE9EBEFA35EEE61BF7B7D041B8EC34C6C00C0C0671C4AE063318FB75BE87AF4FE859608C95F0AB4774F8C95BB

result *y* =
130F8F8B5E1ABB4DD94F6BAAF654A2D5810411E77B7423965E0C7FD79EC1AE563C207BD255EE9828EB7A03FED565240D2CC80DDD2CECBB2EB50F0951F75AD87977F

Appendix P. Test Vector Type B

scalar *k* =
7616133442038E27357DB450C353BD11FBA3BCAC8B7B8C3EF76AADB5FE05BE1
DD57A22D42A5444D00DCD018D389170C54FE781CB21C36020F657D001E1CBB41
DD1

affine coordinate *x* =
BBECF65446053080CC1CF955938C58EB630C84ECAD2756F93B47EBFA9F9BCA3
FA8343539812608CAB2D3A9F8079AB8311A4F269B0A3CD9E0DDD066FC4121D9
2F0E

affine coordinate *y* =
1DD96DB411AD67997B10D42C76B8510C8A930DFA9A5927AC274B0C5021798690
777B8E77E6AE2648BF513E02F586898E7DAE20D71D19838A9F3175F06B057C5F2
F4

The results of function $k(x, y)$ are:

result *x* =
1CE3631976395AD8957F367446D6C99308D5B9E8E0C42DE27CA568CFBE6155D01
6F54AF8A4B751F75AA61255FE09340A8F36A5BD61FD45E0A217123362A459D78
A5

result *y* =
D5AD0E3B4B1BA4C9C462DF92A198067CD4E3176D8F6C710D50B109B3590F7A8
0BCA504D19A2BFAD400713ED774A629EFB6DA24ABB037EFCF4B6040C92BDB4
CAB8D

Appendix Q. Test Vector Type B

Table 71. Test Vector Type C, Modulus *m*.

Format	$m = 2^{521} - 1$, $\log_2(m+1) = 521$
Hexadecimal	000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
Decimal	6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454554977296311391480858037121987999716643812574028291115057151
Number of Bits	521
Is *m* Prime?	Probably

Table 72. Test Vector Type C, Vector *x*.

Format	$x = 2^{544} - 1$
Hexadecimal	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE
Decimal	5758609657015291369997489289838056779353212311426453290368967132943152103259504474008372078212980297151898765610906745757706580551032703601930899431507409734572441414
Number of Bits	544
Is *m* Prime?	Not a Prime

Appendix R. Computing Platform Type A

A computing platform of type A was used for real-time measurements. The following figures describe the operating characteristics of this particular computing platform with respect to its CPU type, memory, and running processes in the system at the times of measurement.

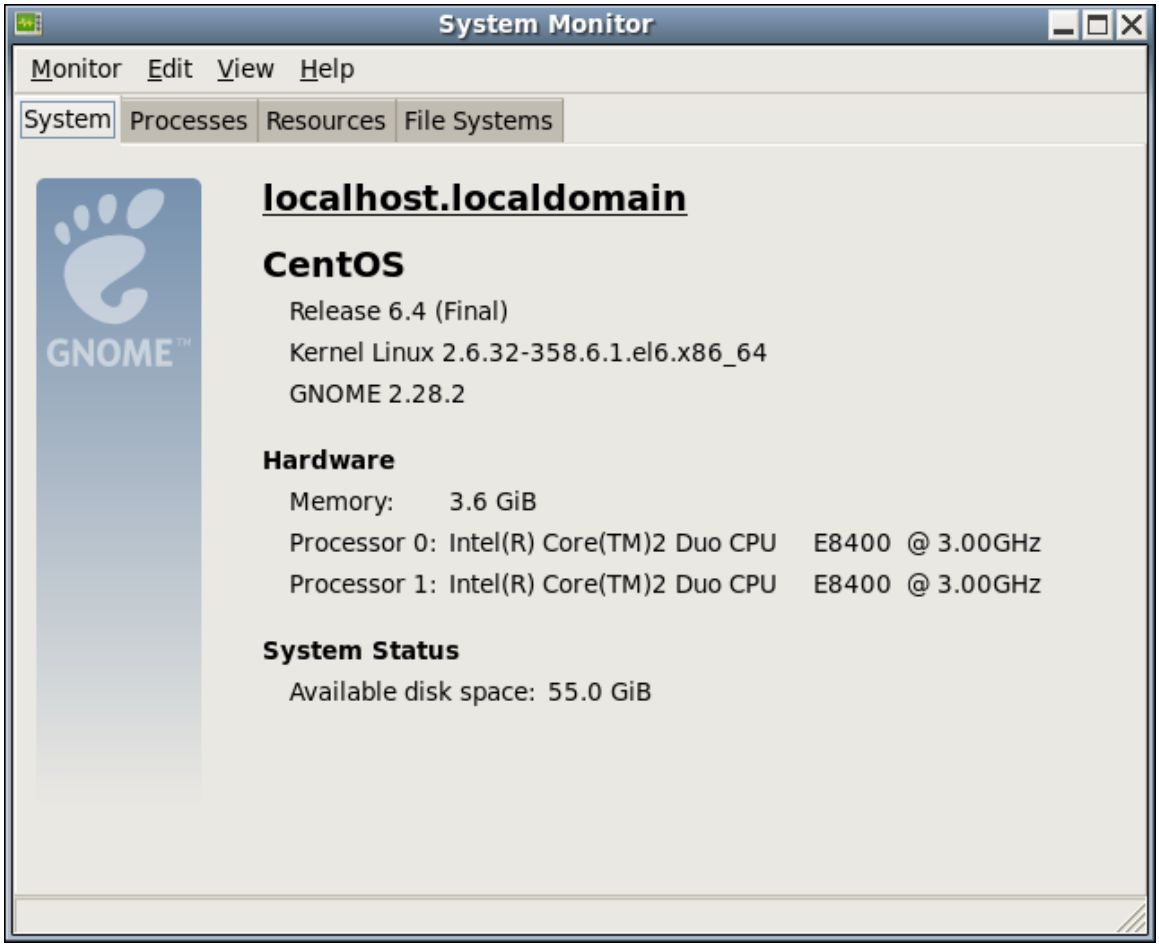

Figure 46. Computing Platform Type A, CPU and Memory

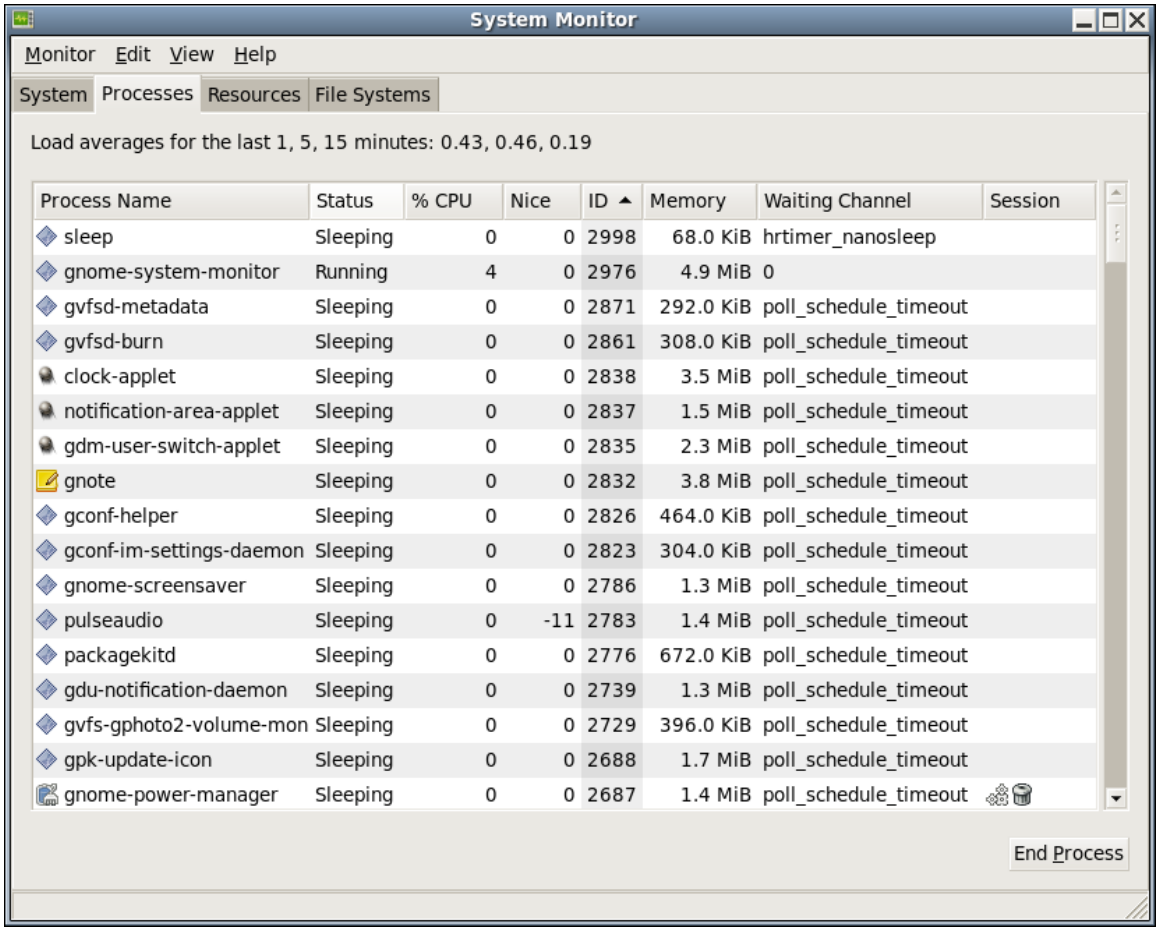

Figure 47. Computing Platform Type A, Running/Sleeping Processes

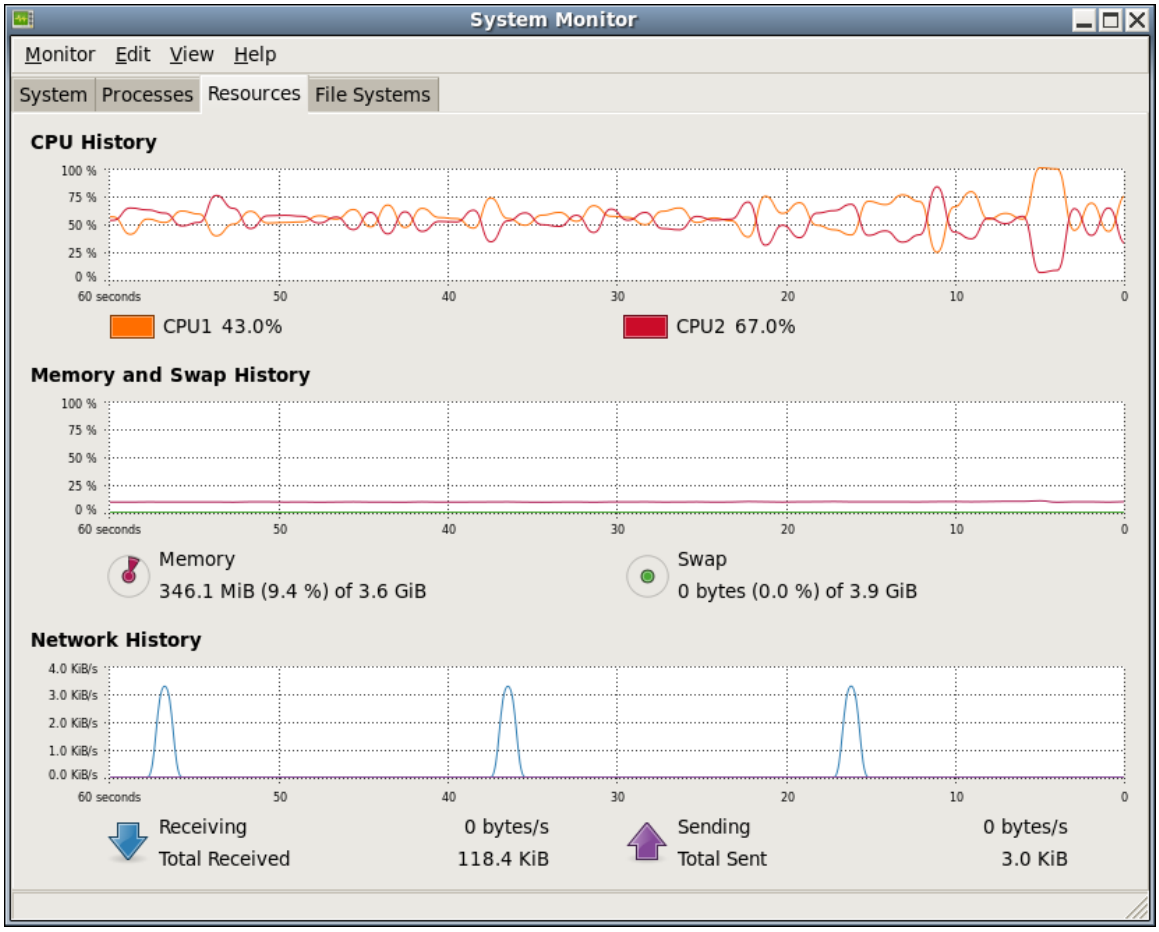

Figure 48. Computing Platform Type A at Busy State

Appendix S. Computing Platform Type B

The following computing platform of type B was applied for real-time measurements:

Figure 49. Computing Platform Type B, CPU and Memory

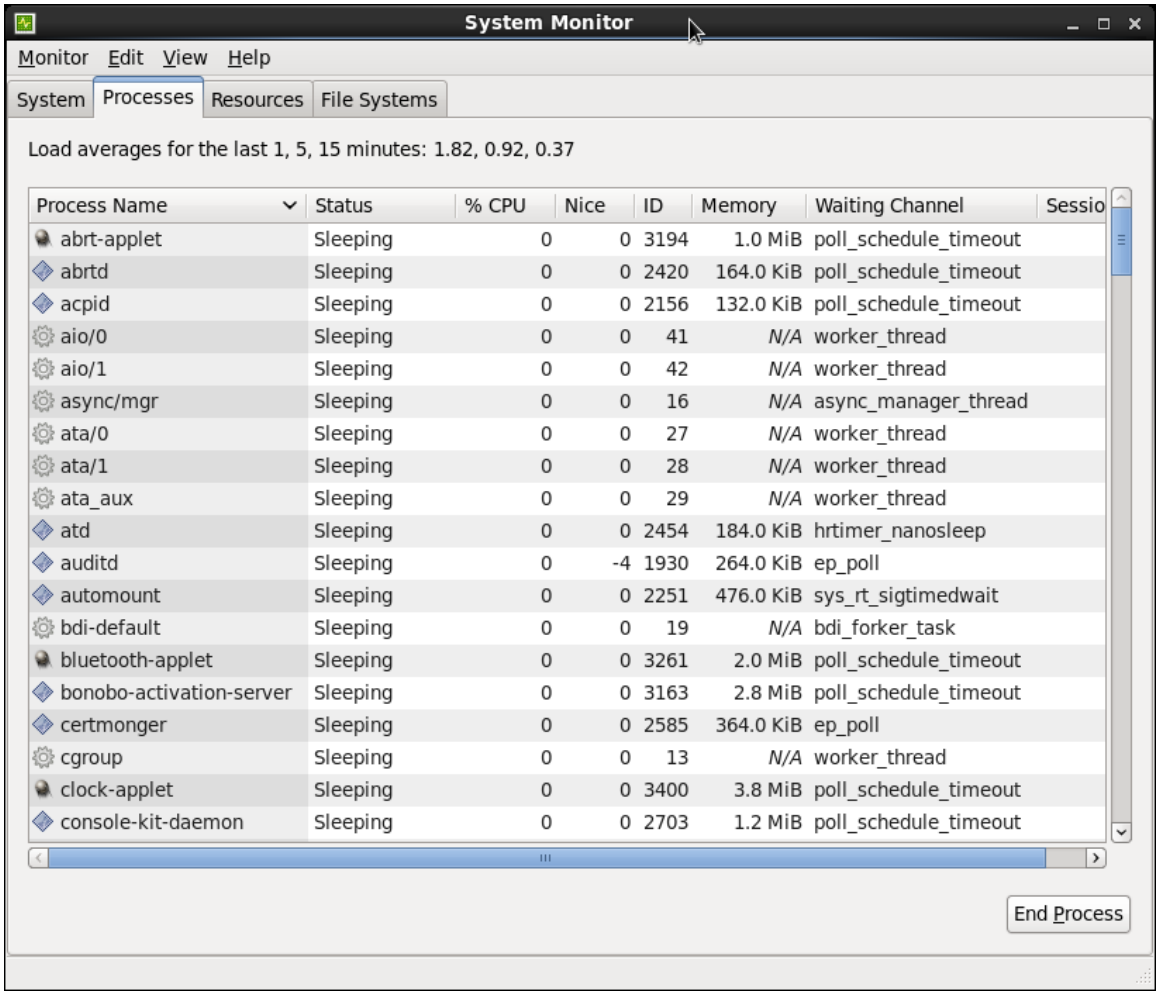

Figure 50. Computing Platform Type B with Running/Sleeping Processes

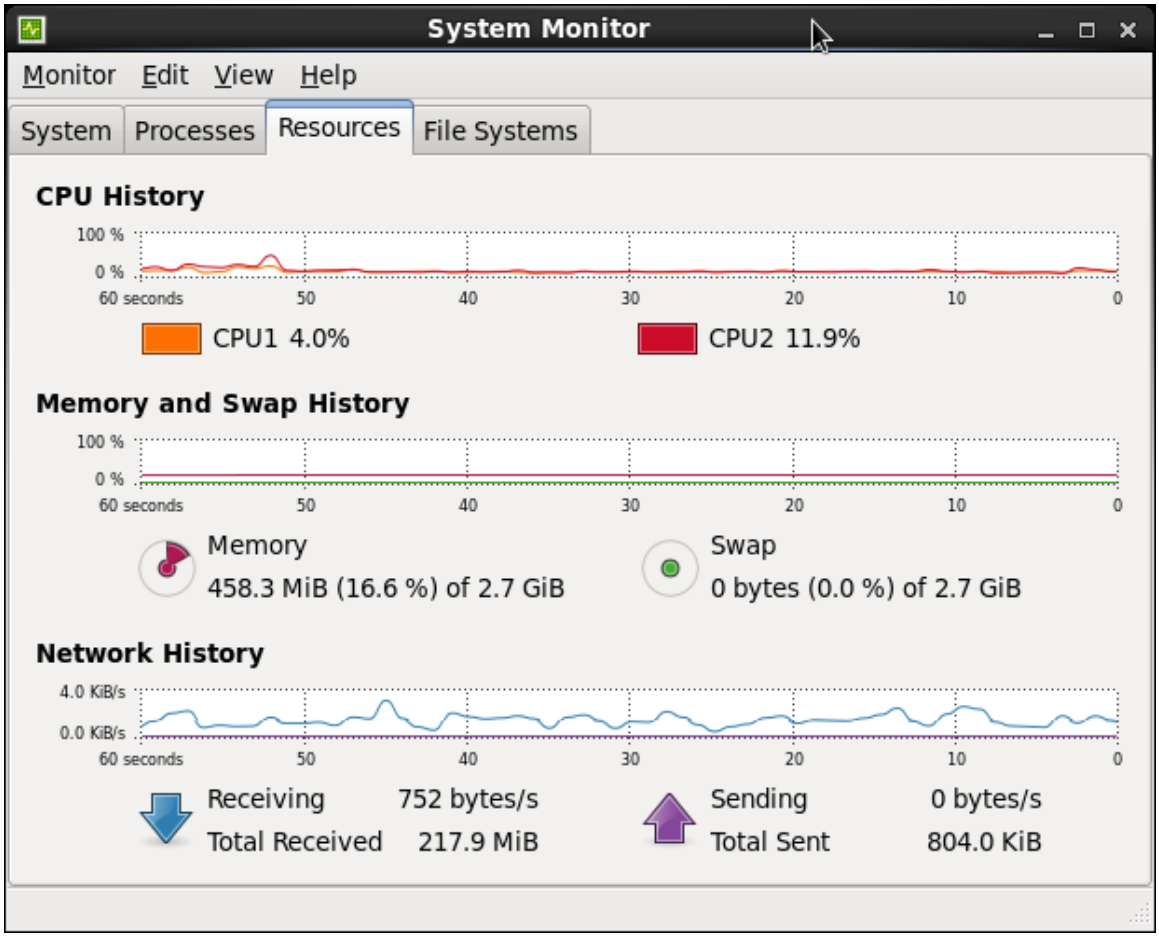

Figure 51. Computing Platform Type B, Resource Utilization

Appendix T. Computing Platform Type C, CPU Resource Busy

A computing platform type C was used for some real-time measurements. Most of CPU resources were allocated to other running processes at the times of measurement. The following figure illustrates the characteristics of this particular computing platform with respect to its CPU type, memory, and running processes in the system at the times of measurement.

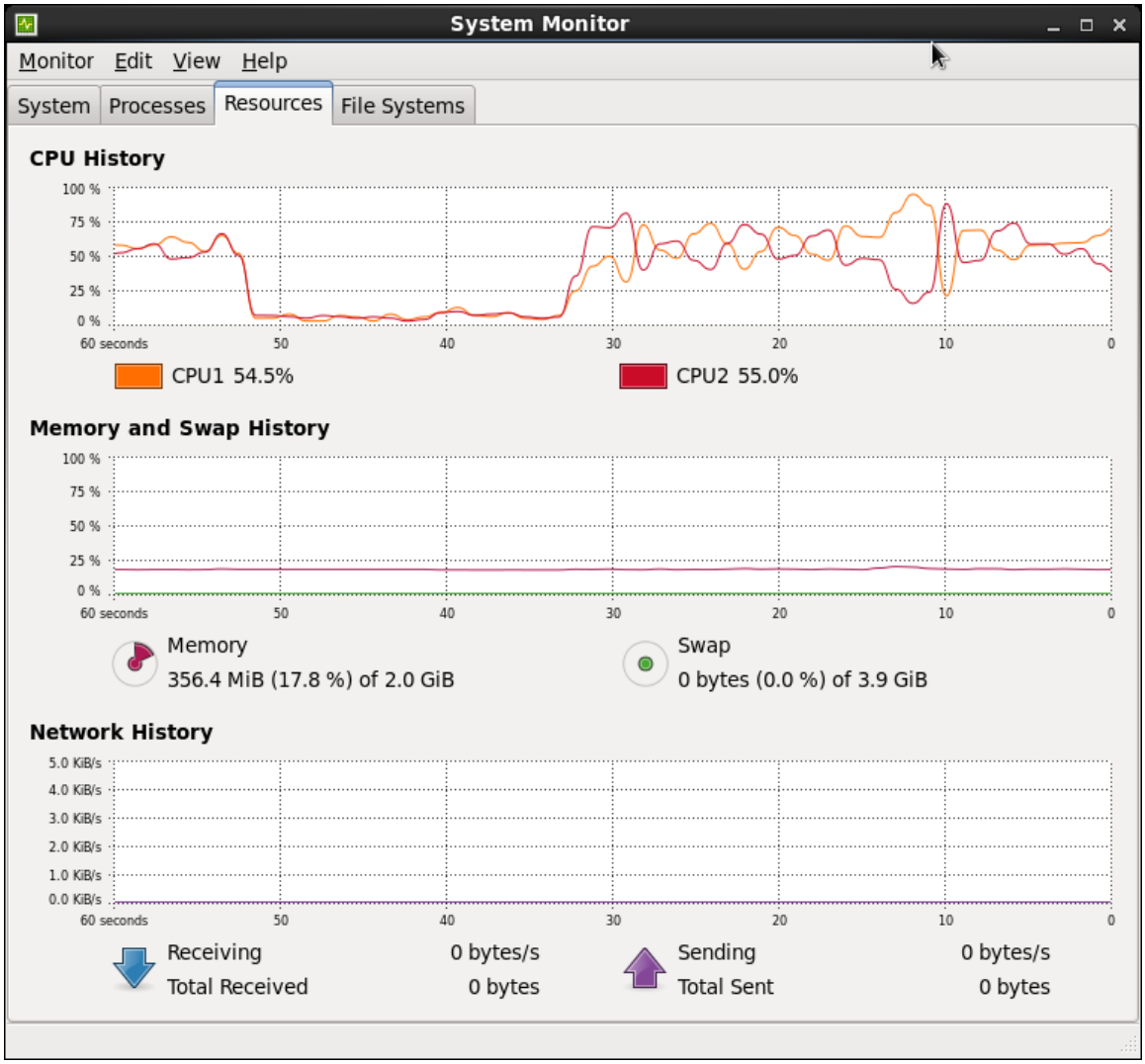

Figure 52. Computing Platform Type C, Resources Busy

Appendix U. Description of Metrics TOT_CYC and TOT_INS

The metric TOT_CYC measures total number of CPU cycles to complete a software function. The TOT_CYC metric is just a convenience way to name the total CPU clock cycles for a program as described generally in (Patterson & Hennessy, 2012, pp. 30-39). In this research, the TOT_CYC was specifically used to measure the total number of CPU clock cycles to accomplish a top-level mathematical function $k(x, y)$ with a fairly known run-time environment (this specific function $k(x, y)$ is to calculate a scalar multiplication with two affine coordinates x and y). Additionally, this research also acquired and analyzed the TOT_CYC metric to assess the CPU clock cycles of sub-modules such as point-adding, or point doubling. Thus, to realize the performance comparison accurately, the measurement pairs: TOT_CYC for NSS and TOT_CYC for OpenSSL must be acquired on the same computing platform, and on the same runtime environment. Then the comparison can be done with each of these measurement pairs.

According to Patterson and Hennessy, the definition of time is called wall clock time, response time, or elapsed time. These terms mean the whole time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead etc. Thus, the metric TOT_CYC acquired by PAPI measurement method can only be an approximation of the entire time to complete a task.

The total number CPU instructions in a software function, namely as metric TOT_INS can be accurately converted to the TOT_CYC according to the computing clock cycle-time per CPU instruction (see Intel Latency, 2013). Consequently, the number of clock cycles required for function $k(x, y)$, or for computing sub-modules can

be written as:

TOT_CYC = *Total Instructions for k*(*x*, *y*) × *Clock Cycles per Instruction* (see Patterson & Hennessy, 2012, p. 33).

From the equation above, one must obtain both coefficients *Total Instructions* and *Clock Cycles per Instruction* in order to derive accurately the TOT_CYC. The TOT_INS metric used in this research is a convenience way to name the *Total Instructions* for a program as described generally in (Patterson & Hennessy, 2012, pp. 30-39).

For example, when the following 64-bit multiplication routine

```
int64_t mul_low_64x64 (int64_t a64, int64_t b64) {
return (int64_t)((__int128_t)a64 * b64);
}
```

is compiled with compiler optimization option 1,

```
gcc -O1 -S D.c  -oD.asm_opti
```

the GCC compiler will produce the following assembly codes

```
   .file   "D.c"
   .text
.globl mul_low_64x64
   .type   mul_low_64x64, @function
mul_low_64x64:
.LFB37:
   .cfi_startproc
   movq   %rsi, %rax
   imulq   %rdi, %rax
   ret
   .cfi_endproc
```

Thus, BOCHS virtual machine will count exactly one "movq" CPU instruction, one "imulq", and one "ret" for the arithmetic routine mul_low_64x64 (int64_t a64, int64_t b64). In that case, the *Total Instructions* coefficient, TOT_INS, must exactly equal to 3.

Since the value of TOT_INS exactly equals to 3, then the following equality must be true for the function mul_low_64x64():

$$\text{TOT\_INS} = \text{MODULE\_COST} = k_1(\text{imulq}) + k_2(\text{movq}) + \text{OHF}$$

where $k_1 = 1$, $k_2 = 1$, and the Overhead Factor, OHF = 1 for the "ret" instruction.

One could convert the TOT_INS to TOT_CYC by referencing the CPU instruction latency of the target computing platform (see Appendix V for the details of acquiring coefficients of TOT_CYC).

Note:

The CPU mnemonic MULq used in this research is a representation of imulq instruction. Thus, the imulq CPU instruction could be a subset of MULq.

The CPU mnemonic MOVq used in this research is a representation of movq instruction. Thus, the movq CPU instruction could be a subset of MOVq.

Appendix V. Description of Metrics imulq and movq

The CPU instruction imulq[8] in an x86, 64-bit hardware platform computes an integer multiplication of two 64-bit operands. The 128-bit result will be stored into two 64-bit registers. In case of the routine

mul_low_64x64 (int64_t a64, int64_t b64),

the target CPU multiplies the content of register RDI (routine parameter a64) to the content of register RAX (routine parameter b64)

imulq %rdi, %rax (machine codes 0x48, 0xF7, 0xEE)

and returns an 128-bit result in a resister pair RDX:RAX, where the register RAX is designated as low-word of the result.

According to Intel literature (see IA-64-32, 2013), and in a summary paper from (Granlund, 2014), executing the MULq instruction for Intel Pentium P4 processor will take exactly ten clock cycles. However, for the Intel Nehalem processors, executing the MULq instruction will take exactly three clock cycles.

The CPU instruction movq[9] in an x86, 64-bit hardware platform moves the data between two 64-bit operands. In case of executing the instruction movq with two CPU registers (no external memory, or cache)

movq %rsi, %rax (machine codes 0x48, 0x89, 0xF8),

it will take one clock cycle for most of processors (P4, AMK K10 etc.)

---

[8] The CPU mnemonic MULq used in this research is a representation of imulq instruction. Thus, the imulq CPU instruction could be a subset of MULq.

[9] The CPU mnemonic MOVq used in this research is a representation of movq instruction. Thus, the movq CPU instruction could be a subset of MOVq.

References

ANSI. (2001). Public Key Cryptography for the Financial Services Industry, Key Agreement and Key Transport Using Elliptic Curve Cryptography, ANSI X9.63.

ANSI. (2005). Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). ANSI X9.62.

ARM. (2013). *ARM Processor Architecture*. The Architecture for Digital World. Retrieved November, 01, 2013 from http://www.arm.com/products/processors/instruction-set-architectures/index.php

Avanzi, R., & Sica, F. (2006). *Scalar Multiplication on Koblitz Curves using Double Bases*. Technical Report Number 2006/067, Cryptology ePrint Archive.

Avanzi, R. (2004). *A Note on the Signed Sliding Window Integer Recoding and a Left-to-Right Analogue*. LNCS vol. 3357, pp. 130–143, Springer, Heidelberg. Retrieved November, 01, 2013 from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.1946&rep=rep1&type=pdf

Aoki, K., Hoshino, F. Kobayashi, & T. Oguro, H. (2001). *Elliptic Curve Arithmetic Using SIMD*. ISC2001, vol. 2200 of Lecture Notes in Computer Science, pp. 235-247, Springer-Verlag.

Afreen, R., Mehrotra, S. C., & Patrick, T. (2011). *A Review on Elliptic Curve Cryptography for Embedded System*. International Journal of Computer Science & Information Technology (IJCSIT), 3(3), June 2011. Retrieved November, 01, 2013 from http://airccse.org/journal/jcsit/0611csit07.pdf

Aigner, H., Bock, H., Hütter, M., & Wolkerstorfer, J. (2004). *A Low-Cost ECC Coprocessor for Smartcards.* Cryptographic Hardware and Embedded Systems - CHES 2004. Lecture Notes in Computer Science, vol. 3156, pp. 107-118, Springer Heidelberg. Retrieved February, 01, 2013 from http://link.springer.com/chapter/10.1007%2F978-3-540-28632-5_8 http://www.iacr.org/archive/ches2004/31560104/31560104.pdf

Barker, E., Barker W., Burr, W., Polk, W., & Smid, M. (2012). *Recommendation for Key Management – Part 1: General (Revision 3).* NIST Special Publication 800-57, pp. 63-64.
Retrieved November, 01, 2013 from
http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf

Brown, M., Hankerson, D., Lopez, J., & Menezes, A. (2001). *Software Implementation of theNIST Elliptic Curves Over Prime Fields*.
Retrieved August, 17, 2013 from
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.6300&rep=rep1&type=pdf

Blake, I., Seroussi, G., & Smart, N. (2001). *Elliptic curves in Cryptology*. Cambridge: Cambridge University Press.

Burton, K., & Yin, Y. (2006). *Storage-Efficient Finite Field Basis Conversion*. RSA Laboratory. Selected Areas in Cryptography '98 Proceedings, Lecture Notes in Computer Science, Springer, 1999, vol. 1556, pp. 81-93.

Bernstein, D., & Lange, T. (2007). *Analysis and Optimization of Elliptic-curve Single-scalar Multiplication. American Mathematical Society*. Contemporary Mathematics, Finite Fields and Applications, vol. 461, pp. 1-19.

BOCHS. (2013). *The Cross Platform IA-32 Emulator*. Bochs 2.6.2 released on May 26, 2013.
Retrieved October, 20, 2013 from
http://bochs.sourceforge.net/

Bos, J., Marcelo, E. Kaihara, K., Thorsten, K., Arjen, K., Lenstra, A., & Montgomery, P. (2009). *On the Security of 1024-bit RSA and 160-bit Elliptic Curve Cryptography*. Alcatel-Lucent Bell Laboratories and Microsoft Research.
Retrieved October, 20, 2013 from
http://eprint.iacr.org/2009/389.pdf

Bos, J., Costello, C., Longa, P., & Naehrig, M. (2014). *Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis*.
Retrieved October, 20, 2013 from
http://research.microsoft.com/pubs/209303/curves.pdf

Bartolini, S., Branovic, I., Giorgi, R., & Martinelli, E. (2008). *Effects of Instruction-Set Extensions on an Embedded Processor: A Case Study on Elliptic-Curve Cryptography over GF(2^m)*. IEEE Transaction on Computer, 57(5), May 2008.
Retrieved October, 20, 2013 from
http://www.dii.unisi.it/~giorgi/papers/Bartolini08a.pdf

Bennett, M. K. (1995). *Affine and Projective Geometry*. New York: John Wiley & Sons Inc.

Bi, G., & Zeng, Y. (2004). *Transforms and Fast Algorithms for Signal Analysis and Presentations*. p. 27. Boston: Birkhauser.

BBOD. (2013). *Black-Box Optimization Benchmarking*. Retrieved January, 07, 2014 from http://coco.gforge.inria.fr/doku.php?id=bbob-2013

CAVP. (2013). *Cryptographic Algorithm Verification Program*. NIST Computer Security Division. Computer Security Resources Center. Retrieved January, 09, 2013 from http://csrc.nist.gov/groups/STM/cavp/

CAVP LABS. (2014). *Testing Laboratories.* Cryptographic Algorithm Verification Program. NIST Computer Security Division. Computer Security Resources Center. Retrieved May, 09, 2014 from http://csrc.nist.gov/groups/STM/testing_labs/index.html

CAVP OpenSSL. (2012). *Validated FIPS 140-1 and FIPS 140-2 Cryptographic Modules.* Cryptographic Algorithm Verification Program. NIST Computer Security Division. Computer Security Resources Center. Retrieved May, 09, 2014 from http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/1401val2012.htm

CAVP NSS. (2010). *Validated FIPS 140-1 and FIPS 140-2 Cryptographic Modules.* Cryptographic Algorithm Verification Program. NIST Computer Security Division. Computer Security Resources Center. Retrieved May, 09, 2014 from http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/1401val2010.htm

Ciet, M., Joye, M., Lauter, K., & Montgomery, P. (2006). *Trading Inversions for Multiplications in Elliptic Curve Cryptography.* Designs, Codes and Cryptography, 39(2), pp. 189-206. Retrieved August, 17, 2013 from http://research.microsoft.com/apps/pubs/default.aspx?id=178808

Certicom Research. (2009). *Elliptic curve Cryptography*. Retrieved August, 17, 2013 from www.secg.org/download/aid-780/sec1-v2.pdf

Certicom Research. (2004). *An Elliptic Curve Cryptography (ECC) Primer.* Retrieved August, 17, 2013 from http://www.certicom.com/images/pdfs/WP-ECCprimer.pdf

Certicom Research. (1999). *GEC 2: Test Vectors for SEC 1*.
Retrieved August, 17, 2013 from
http://www.secg.org/download/aid-390/gec2.pdf
http://www.secg.org/

Cantor, D. (1987). *Computing in the Jacobian of a Hyperelliptic Curve*.
Mathematics of Computation, 48(177), pp. 95-101.
Retrieved August, 17, 2013 from
http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866101-0/S0025-5718-1987-0866101-0.pdf

Cohen, H., Miyaji, A., & Ono, T. (1998). *Efficient Elliptic Curve Exponentiation Using Mixed Coordinates.* Advances in Cryptology - ASIACRYPT 98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1514, pp. 51-65, Springer, 1998.
Retrieved August, 17, 2013 from
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.2566

Cohen, H., Frey, G., Doche, C., & Avanzi, R. (2006). *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Boca Raton, FL: Chapman & Hall/CRC.

Cohn, H. (1962). *Advanced Number Theory*. New York: Dover Publications.

Connel, I. (1999). *Elliptic Curve Handbook.* McGill University.

Code XL. (2013). *Advance Micro Devices Development Central*.
Retrieved October, 20, 2013 from
http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/

Crypto++. (2007). *Crypto++® Library 5.6.2*.
Retrieved October, 20, 2013 from
http://www.cryptopp.com/benchmarks-p4.html

COCO. (2013). *Comparing Continuous Optimisers.*
Retrieved March, 20, 2014 from
http://coco.gforge.inria.fr/doku.php

CST. (2014). *Cryptographic and Security Testing.* Directory of Accredited Laboratories, National Volunteer Laboratory Accreditation Program.
Retrieved April, 09, 2014 from
http://ts.nist.gov/standards/scopes/crypt.htm

CSE, (2014). *Communications Security Establishment.* Government of Canada. Retrieved July, 06, 2014 from http://www.cse-cst.gc.ca/index-eng.html

Mihocka, D., & Shwartsman, S. (2014). *Virtualization Without Direct Execution or Jitting:Designing a Portable Virtual Machine Infrastructure.* Retrieved January, 20, 2014 from http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf

Drongowski, D. (2008). *Basic Performance Measurements for AMD Athlon™ 64, AMD Opteron™ and AMD Phenom™ Processors. Advnace Micro Devices*. Retrieved October, 20, 2013 from http://developer.amd.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf

EFD. (2001). *Explicit-Formulas Database*. Retrieved August, 17, 2013 from http://hyperelliptic.org/EFD

EFD_Double. (2001). *Explicit-Formulas Database*. Retrieved August, 17, 2013 from http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-3.html#doubling-dbl-2001-b

EFD_Add. (2007). *Explicit-Formulas Database*. Retrieved August, 17, 2013 from http://hyperelliptic.org/EFD/g1p/auto-shortw-jacobian-3.html#addition-add-2007-bl

eBACS. (2004). *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. European Network of Excellence in Cryptology II Retrieved October, 20, 2013 from http://bench.cr.yp.to/

FIPS-140-1. (1994). *Security Requirements for Cryptographic Modules*. FIPS PUB 140-1. Retrieved July, 06, 2014 from http://csrc.nist.gov/publications/fips/fips1401.htm

FIPS-140-2. (2001). *Security Requirements for Cryptographic Modules*. FIPS PUB 140-2. Retrieved August, 17, 2013 from http://csrc.nist.gov/groups/STM/cmvp/standards.html http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf

FIPS-197. (2001). *Specification for the Advanced Encryption Standard*. FIPS PUB 197. Retrieved August, 17, 2013 from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

FIPS PUB 186-4. (2013). *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication, p. 87.
Retrieved January, 01, 2014 from
http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

FIPS-1747. (2014). *Validated FIPS 140-1 and FIPS 140-2 Cryptographic Modules*.
Retrieved July, 06, 2014 from
http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140val-all.htm#1747

Fong, K., Hankerson, D., Lopez, & J., Menezes, A. (2004). *Field Inversion and Point Halving Revisited*. IEEE Trans. Computers, vol. 53, p. 1047–1059.

Jennic JN5148. (2010). *Jennic Wireless Mote*.
Retrieved October, 20, 2013 from
http://www.jennic.com/products/wireless_microcontrollers/jn5148

Jones, C. (2010). *Software Engineering Best Practices. Lessons from Successful Projects in the Top Companies.* NY: McGraw Hill.

Floyd, R. (1967). *Nondeterministic Algorithms*. Journal of the ACM, 14(4), p. 11, pp. 640-642.
Retrieved November, 01, 2013 from
http://repository.cmu.edu/cgi/viewcontent.cgi?article=2787&context=compsci

Fagan, M. (1976). *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal 15(3), pp. 182-211, 1976.

Greenberg, M. (1995). *Euclidean and non-Euclidean Geometries. Development and History*, 3rd Ed. New York: W.H Freeman.

Gillham, B. (2003). *Case Study Research Methods*. Series Continuum Research Methods, 1st Ed. Bloomsbury Academic, September 13, 2000.

Gilb, T. & Graham, D. (1993). *Software Inspection*. MA: Addison Wesley.

GNU-MP. (2011). *The GNU Multiple Precision Arithmetic Library*, 5.0.2. Ed., p. 116.
Retrieved November, 01, 2013 from
http://gmplib.org/

GNU-CPU-Time. (2014). *CPU Time Inquiry.*
Retrieved March, 01, 2014 from
http://www.gnu.org/software/libc/manual/html_node/CPU-Time.html

GCC. (2013). *The GNU Compiler Collection*.
Retrieved November, 01, 2013 from
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
http://gcc.gnu.org/

Gell-Mann, M. (1995). *What is complexity?* Complexity 1(1): pp. 16-19.

Granlund, T. (2014). *Instruction Latencies and Throughput forAMD and Intel x86 Processors.*
Retrieved August, 09, 2014 from
http://gmplib.org/~tege/x86-timing.pdf

Hennessy, J., & Patterson, D. (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufman.

Hankerson, D., Menezes, A., & Vanstone, S. (2004). *Guide to Elliptic Curve Cryptography*. NY: Springer-Verlag Inc.

Herrmann, D. (2007). *Complete Guide to Security and Privacy Metrics: Measuring Regulatory Compliance, Operational Resilience, and ROI*. Boca Raton FL: Auerbach Publications.

Itoh, T., Tsujii, S. (1988). *A Fast Algorithm for Computing Multiplicative Inverses in GF(2^n) using Normal Bases*. Information and Computation, vol. 78, pp.171-177.

IA-64-32. (2013). *Intel 64 and IA-32 Architectures Software Developer's Manual*, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Section 4.2.
Retrieved November, 01, 2013 from
http://download.intel.com/products/processor/manual/325462.pdf

IA-64. (2013). *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*. Order Number: 253669-047 US. June 2013.
Retrieved November, 01, 2013 from
http://download.intel.com/products/processor/manual/253669.pdf

Intel AVX. (2011). *Using Intel Advanced Vector Extensions to Implement an Inverse Discrete Cosine Transform.*
Retrieved November, 01, 2013 from
http://software.intel.com/en-us/articles/using-intel-advanced-vector-extensions-to-implement-an-inverse-discrete-cosine-transform/

IASE. (2013). *Information Assurance Support Environment*. Public Key Infrastructure (PKI) and Public Key Enabling (PKE). Retrieved from http://iase.disa.mil/pki-pke/index.html http://iase.disa.mil/index2.html

IEC-14756. (1999). *Measurement and Rating of Performance of Computer-based Software Systems.*

IEEE 1363. (2000). *Standard Specifications for Public-Key Cryptography*. IEEE Std 1363.12.2000 (Rev: 2000).

IEEE 610-12. (2002). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12.1990 (Rev: 2002).

IEEE 982.1. (1998). *Standard Dictionary of Measures to Produce Reliable Software*. IEEE Std 982.1-1988.

IEEE 982.2. (1998). *Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software*. IEEE Std 982.2-1988.

IEEE 982.1. (2005). *Standard Dictionary of Measures of the Software Aspects of Dependability*. IEEE Std 982.1-2005.

IEEE 1028. (1997). *Standard for Software Reviews*. IEEE Std 1028-1997.

IEEE 1028. (2008). *Standard for Software Reviews and Audits*. IEEE Std 1028-2008.

Intel PERC. (2013). *Performance Counter Monitor. Intel® Performance Counter Monitor - A better way to measure CPU utilization*. Retrieved October, 20, 2013 from http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization

Intel Xscale. (2007). *3rd Generation Intel XScale® Microarchitecture*. Developer Manual. May 2007. Retrieved November, 01, 2013 from http://download.intel.com/design/intelxscale/31628302.pdf

Intel Latency. (2013). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Order Number 248966-028, p. C-21, C-26, July 2013. Retrieved November, 01, 2013 from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

ISO/IEC 15939. (2001). *Software Engineering - Software Measurement Process.*

ISBSG. (2007). *ISBSG R10 Database*. International Software Benchmarking Standards Group.

ISBSG. (2006). *Glossary of Terms*, version 5.9.1. International Software Benchmarking Standards Group.

ITU-X509. (2012). *X.509: Information Technology - Open Systems Interconnection - The Directory: Public-key and Attribute Certificate Frameworks.* Retrieved January , 2014 from http://www.itu.int/rec/T-REC-X.509-201210-I/en http://www.itu.int/rec/T-REC-X.509

Joye, M. (2008). *Fast Point Multiplication on Elliptic Curves without Precomputation. Arithmetic of Finite Fields*. Lecture Notes in Computer Science, vol. 5130, pp. 36-46. Berlin Heidelberg: Springer-Verlag. Retrieved November, 01, 2013 from http://link.springer.com/chapter/10.1007%2F978-3-540-69499-1_4 http://www.joye.site88.net/papers/Joy08fastecc.pdf

Kasper, E. (2012). *Fast Elliptic Curve Cryptography in OpenSSL*. Belgium, ESAT/COSIC, Katholieke Universiteit Leuven, Belgium Proceeding FC'11 Proceedings of the 2011 International Conference on Financial Cryptography and Data Security, pp. 27-39. Heidelberg, Berlin: Springer-Verlag. Retrieved October, 20, 2013 from http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/37376.pdf

Koc, C. (2009). *Cryptographic Engineering*. Springer Science Business Media, LLC.

Koblitz, N. (2000). *Efficient Arithmetic on Koblitz Curves*. Designs, Codes and Cryptography - Special issue on towards a quarter-century of public key cryptography, 19(2-3), pp. 195-249. Norwell, MA: Kluwer Academic Publishers. Retrieved October, 20, 2013 from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.157.2469 http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=0F99ACB6138C89D0591A157C1B73E55D?doi=10.1.1.157.2469&rep=rep1&type=pdf

Kepler, J. (1571). *Johannes Kepler: His Life, His Laws and Times*. Retrieved October, 20, 2013 from http://kepler.nasa.gov/Mission/JohannesKepler/

Knuth, D. (1969). *The Art of Computer Programming, vol. II: Seminumerical Algorithm,* . exercises 6 and 7, p. 7. Addison-Wesley.

Keyes, J. (2005). *Software Engineering Handbook*. Boca Raton NY: Auerbach Publications.

Levinthal, D. (2009). *Performance Analysis Guide for Intel Core™ i7 Processor and Intel® Xeon™ 5500 Processors*. Retrieved October, 20, 2013 from http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Lathi, P. (1998). *Signal Processing and Linear Systems*, p. 641. Berkerley-Cambrigde.

Li, T. (2008). *Digital Signal Processing Fundamentals and Applications*, p. 87. Elsevier.

Laird, L., & Brennan, M. (2006). *Software Measurement and Estimation: A Practical Approach*. John Wiley & Sons Inc.

Menezes, A., Oorschot, P., & Vanstone, S. (1996). *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press LLC.

Mittelmann, H., & Pruessner, A. (2004). *A Server for Automated Performance Analysis of Benchmarking Data*. Arizona State University. Retrieved October, 20, 2013 from http://plato.asu.edu/ftp/papers/paper99.pdf

MIRACL. (2013). *Multiprecision Integer and Rational Arithmetic*. Retrieved November, 01, 2013 from http://www.certivox.com/miracl/

Maeder, R. (1996). *Long Integers: Efficient Algorithm*. Mathematica Programmer, 6(3), Summer 1996. Retrieved March, 01, 2014 from http://www.mathematica-journal.com/issue/v6i3/columns/maeder/contents/63maeder.pdf

Morowitz, H. (2002). *The Emergence of Everything*: *How the World Became Complex*. Newyork, NY: Oxford University Press.

NSS. (2013). *Network Security Services*. Retrieved August, 17, 2013 from http://developer.mozilla.org/en-US/docs/NSS https://developer.mozilla.org/en-US/docs/NSS_Sources_Building_Testing

NSS-1. (2013). *Overview of NSS. Open Source Crypto Libraries.* Retrieved August, 17, 2013 from http://developer.mozilla.org/en-US/docs/Overview_of_NSS

NSS-2. (2014). *NSS Open Source Repository*
Retrieved March, 30, 2014 from
ftp://ftp.mozilla.org/pub/mozilla.org/security/nss/releases/NSS_3_16_RTM/src/

NIST 800-56A. (2013). *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography.* NIST Special Publication 800-56A, Revision 2.
Retrieved January, 17, 2014 from
http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf

NIST. (2010). *Mathematical Routines for the NIST Prime Elliptic Curves*.
Retrieved August, 17, 2013 from
http://www.nsa.gov/ia/_files/nist-routines.pdf

NIST. (2008). *Performance Measurement Guide for Information Security.* NIST Special Publication 800-55, Revision 1.
Retrieved August, 17, 2013 from
http://csrc.nist.gov/publications/nistpubs/800-55-Rev1/SP800-55-rev1.pdf

NIST. (2007). *Key Exchange Establishment Elliptic Curve Diffie Hellman (ECDH) Using 256 and 384-bit Prime Moduli*. NIST Special Publication 800-56A.
Retrieved August, 17, 2013 from
http://csrc.nist.gov/groups/ST/toolkit/documents/SP800-56Arev1_3-8-07.pdf

NIST. (2003). *Security Metrics for Information Technology Systems*. National Institute of Standards and Technology Special Publication 800-55, July 2003.

NSA. (2013). *NSA Suite B Cryptography*. National Security Agency.
Retrieved January 21, 2013 from
http://www.nsa.gov/ia/programs/suiteb_cryptography/
http://www.gdc4s.com/Documents/Products/COTS%20Tectical%20Wireless%20Networking/Fortress%20Secure%20Client%20Comms/SuiteB_FINAL.pdf
http://www.gdfortress.com/Technology-Article/suite-b.html

Fenton, N., & Pfleeger, S. (1996). *Software Metrics: A Rigorous and Practical Approach*. (2nd Edition). International Thomson Computer Press.

OpenSSL. (2013). *Open Security Sockets Layer*.
Retrieved August, 17, 2013 from
http://www.openssl.org/
http://www.openssl.org/source/

OpenSSL-1. (2014). *Open Source Repository. Open Security Sockets Layer*.
Retrieved March, 30, 2014 from
https://www.openssl.org/source/

OMB. (2012). *Fiscal year 2011 Report to Congress on the Implementation of the Federal Iinformation Security Management Act of 2002*. Office of Management and Budget Retrieved from http://www.whitehouse.gov/sites/default/files/omb/assets/egov_docs/fy11_fisma.pdf

PAPI. (2013). *Performance Application Programming Interface*. Innovative Computing Laboratory at the University of Tennessee. Retrieved October, 20, 2013 from http://icl.cs.utk.edu/papi

Pollard, M. (1975). *A Monte Carlo method for factorization*. BIT Numerical Mathematics 15(3), pp. 331–334.

Pohlig, S., & Hellman, M. (1978). *An Improved algorithm for computing logarithms over GF(p) and its cryptographic significance*. IEEE transactions on Information Theory 24, pp. 106–110.

Paré, G. (2004). *Investigating Information Systems with Positivist Case Research*. Communications of the Association for Information Systems, 3(18). Retrieved October, 20, 2013 from http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=32CF66873A325F269E0100CA17671D82?doi=10.1.1.65.4667&rep=rep1&type=pdf

Patterson, D. & Hennessy, J. (2012). *Computer Organization and Design, the Hardware/Software Interface*. New York, NY: Elsevier.

Rovenski, V. (2006). *Differential Geometry of Curves and Surfaces*, pp. 65-70. Boston, MA: Birkhauser.

RSA Key Size. (2013). *How Large a Key Should be Used in the RSA Cryptosystem?* RSA Laboratories. Retrieved January 21, 2013 from http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/how-large-a-key-should-be-used.htm

Rosen, K. (2006). *Discrete Mathematics and Its Applications,* 3rd Ed. Boca Raton, FL: Chapman & Hall.

Ryabko, B., & Fionov, A. (2005). *Basics of Contemporary Cryptography for IT Practitioners,* vol. 1. Series on Coding Theory and Cryptology. Hackensack, NJ: World Scientific Publishing Company (September 2005).

Runeson, P., Andersson, C., Thelin, A., Andrews, A. & Berling, T. (2006). *What Do We Know about Defect Detection Methods*. IEEE Software, 23(3), pp. 82-90, May/June 2006.

SEI. (1997). *C4 Software Technology Reference Guide — A Prototype.* Software. Engineering Institute, Carnegie Mellon University. Pittsburgh, Pennsylvania. Retrieved January 21, 2013 from http://www.sei.cmu.edu/reports/97hb001.pdf

Stevenhagen, P. (2008). *The Number Field Sieve*. Algorithmic Number Theory, 44, pp. 83-100. MSRI Publications.

Somani, T. (2010). *Performance Evaluation of Elliptic Curve Projective Coordinates with Parallel GF(p) Field Operations and Side-Channel Atomicity.* Journal of Computers, 5(1), Jan 2010. Retrieved October, 20, 2013 from http://ojs.academypublisher.com/index.php/jcp/article/view/050199109

Solinas, J. (1999). *Generalized Mersenne Numbers*. Tech. Report Centre for Applied Cryptographic Research. Retrieved November, 01, 2013 from http://cacr.uwaterloo.ca/techreports/1999/corr99-39.pdf

SEC 1. (2000). *Standards for Efficient Cryptography Group*. *SEC 1: Elliptic Curve Cryptography*, 1st Ed., Sep. 2000. Retrieved November, 01, 2013 from http://www.secg.org/download/aid-385/sec1_final.pdf http://www.secg.org/index.php?action=secg,docs_secg

Saldamli, G., & Koc, K. (2009). *Cryptographic Engineering*, p. 125. Springer Science.

Szerwinski, R., Guneysu, T. (2008). *Exploiting the Power of GPUs for Asymmetric Cryptography*. Workshop on Cryptographic Hardware and Embedded Systems (CHES'08), LNCS (5154), pp. 79–99.

Source-Selection. (2011). *Source Selection procedures.* Department of Defense. Retrieved July, 01, 2013 from http://www.acq.osd.mil/dpap/policy/policyvault/USA007183-10-DPAP.pdf

Singhal, N., & Raina, J. (2011). *Comparative Analysis of AES and RC4 Algorithms for Better Utilization. International*. Journal of Computer Trends and Technology. July to Aug Issue 2011, pp. 177-181.

Salomon, D. (2006). *Transformations and Projections in Computer Graphics*. pp. 98-101. London: Springer-Verlag.

Shukri, A., Wakid, D., Richard, K., & Dolores, R. W. (1999). *Toward Credible IT Testing and Certification*, pp. 39-47. IEEE Software, July-Aug 1990

US-CERT. (2014). *United States Emergency Readyness Team*. Retrieved January, 01, 2014 from http://www.us-cert.gov/

VirtualBox. (2014). *VirtualBox by Oracle*. Oracle Corporation. Retrieved January, 01, 2014 from https://www.virtualbox.org

Veblan, O., & Bussey, W. (1906). *Finite Projective Geometry.* Retrieved October, 20, 2013 from http://www.ams.org/journals/tran/1906-007-02/S0002-9947-1906-1500747-6/S0002-9947-1906-1500747-6.pdf

WindRiver SIMICS. (2013). *Wind River Simics Full System Simulation.* Retrieved October, 20, 2013 from http://www.windriver.com/products/simics/ http://www.windriver.com/products/simics/simics_po_0520.pdf

Watson, A. H., & McCabe, T. J. (1996). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric* (NIST Special Publication 500-235). Retrieved May, 20, 2014 from http://www.mccabe.com/pdf/mccabe-nist235r.pdf

Yin, K. (2003). *Case Study Research, Design and Methods*, 3rd Ed. Beverly Hills, CA: Sage Publications.

Yin, K. (1994), *Case Study Research. Design and Methods*, 2nd Ed. Thousand Oaks, CA: Sage Publications,

ZigBee. (2010). *ZigBee Alliance, Understanding ZigBee* Retrieved January, 01, 2014 from http://www.zigbee.org/About/UnderstandingZigBee.aspx http://www.zigbee.org/