ABSTRACT

This derivation of symmetric encryption keys using chaotic systems is a result of research in elliptic curve cryptographic protocols and the chaotic wanderings of stars around black holes. This document is an analytical specification of a single method for synchronizing chaotic systems to produce a sequence of encryption keys. The key generator runs in real-time, and its output keys are able to support Cryptographic Block Chaining (CBC) Mode Symmetric Key Cryptography, rotating keys for each block of data encrypted. In the case of a dynamic orbit that results in strong keys, it is possible that the key sequence will never be reused as dynamic chaotic systems produce very non-linear results exhibiting a high degree of entropy. This synchronized chaotic key generator allows encryption keys to be computed from values within an initial handshake protocol and that only a time slice or computational subscript of the actual encryption key be sent using the key transport medium when keys are rotated. This Synchronized Chaotic Key Generator is designed to enable constrained systems to transfer secure high-rate data streams over mixed wired and wireless local area network (LAN) or wireless local area network (WLAN) systems with minimal energy and computational resource consumption as a direct result of key generation.

Table of Contents

LIST OF FIGURES

1. BACKGROUND AND RATIONALE

1.1. History

The interest in network security is as old as the first wired network itself. During the early days of networking, even before the Internet, various networks were attacked by hackers using a method called war dialing. [Ameter 2003] War dialing is performed using one computer to call lots of telephone numbers looking for a computer or remote area dial up server (RADIUS) as an entry point where attacks could be initiated on the target network. The threat still exists today, but hackers do not need to gain access to an inbound telephone line. Instead, they engage in an activity called war driving. The attacker may be sitting in the parking lot or several miles away using a slightly modified 802.11 antennae. Wireless networks provide a whole new set of opportunities for the savvy network hacker. Wireless connectivity is here to stay. It not only boosts productivity, it also provides a large return on investment (ROI) in terms growth potential where wired networks would be difficult to install.

Wired home networks are also a thing of the past as wireless connectivity has improved and the costs of [Wi-Fi 2003] Wi-Fi connectivity have fallen. Few people realize how much information can be stolen and the types of attacks that can be launched from an unprotected wireless network. In large wireless networks, the highest threat comes from rogue access points (AP). A rogue AP is any AP that was not installed by the network manager or his department. [Ameter 2003] Most of these rogue access points are not configured correctly, and sit behind the network firewall, within the trusted network.

Wireless security has been studied at Carnegie Mellon University [Arabasz 2002] since 1994, but after the events of 9/11/2001, security became a critical topic. As a result of their efforts, most wireless networks which include confidentiality, integrity, and availability of the data transmitted to the wired (LAN) network are based on their model. Large wireless networks (WLANs) are configured with centralized access control based on the user's rights on the system. [Ameter 2003, Wi-Fi 2003, Arabasz 2003, Riley 2003, Chou 2005, O'Hara 2005, Johnson 2003] In most cases, a RADIUS (Remote Authentication Dial In User Service) server is employed to authenticate the wireless user on the network just as if the wireless were the same as the wired connection. Many systems administrators have been duped into thinking that their networks are secure when the opposite is actually true. The service set identifier (SSID) and Media Access Control (MAC) address of a networking device, such as the access point (AP), and the Media Access Control address (MAC) of the client computer are transmitted in the clear, meaning no encryption is applied. Any security mechanism based on these values cannot be trusted.

[Ameter 2003] The basic toolkit for a wireless intruder is either a wireless ready computer or a pc connected to its own (rogue) access point, and a wireless packet sniffer, many of which are open source and can be downloaded for free. Attackers will use the same hardware and tools that any network user or administrator will use to gain authorized access and perform administrative functions. The intruder can open packets, and in many cases modify the packet contents to gain access. Cryptography has been employed to protect packet contents from arbitrary inspection and modification. The original wireless standard (802.11) contains a provision for the Wired Equivalency Protocol (WEP). WEP has not only been proven to be insecure because it uses a crippled version of the RC4 encryption algorithm, its use is dangerous to the privacy of data on the network. [Arabasz 2003] Unencrypted data can easily be recovered from the message itself. To make matters worse, the attacker can forge new WEP encrypted packets without ever learning the encryption key.

[Wagner 1997] In 1997, the cellular message encryption algorithm (CMEA) came under intense scrutiny as cellular telephones began the move from a purely open analog architecture to digital network architecture. The problem was cell phone cloning. Attackers were stealing the identity of an authorized user to make free long distance phone calls, and cellular providers were losing large sums of money as a result. Cryptography was chosen as a method of eliminating this threat. However, the method that was chosen did not go through an open review process. Once the cryptography standard was "scanned and posted on the internet by an anonymous party," the standard failed close cryptanalytic scrutiny and could be broken with just 4 known plaintext messages. [Huston 2005] The cellular telephone industry has learned from their mistakes and joined the Internet Engineering Task Force (IETF) in the review process. Both the IETF and OMA (Open Mobile Alliance) publish protocols primarily used in wireless networking. Digital mobile (cellular) telephones are based on a version of the 802.11 standard and may use either IPv4 or IPv6 depending on the network area they are operating in, the capabilities of the client, and other variables.

1.2. Authenticity and Integrity Decisions

The goal of any successful security plan is to prevent unauthorized users from gaining access and disrupting the normal flow of information on the network. Any failure of authentication will cascade into a failure of the integrity of the network and compromise of information resources or databases that are accessible over the network. Even authorized users must be limited from accessing networked resources. In the case of a university [Johnson 2003], students should not have access to resources that allow them to change their grades or modify information stored on the university's human resources network. Users must be limited to only those resources they are authorized to access. The 802.11b standard treated all users alike. Role based access operating at IP stack protocol layer 3, one layer above 802.11b, was needed to provide centralized control and network integrity. [Johnson 2003] Simon Fraser University chose to use a proprietary solution that exceeded their original goals. They were methodical and examined all of the options before choosing a company to install their wireless network. There have been many different proprietary solutions over the decade between 1993 and 2003. Not all companies provided solutions that worked well with solutions provided by others. Basically many hardware components were not interoperable because different manufacturers made trade-offs in manufacturing and software drivers.

The Synchronized Chaotic Key Generator removes integrity verification message fields used by prior solutions. [Riley 2003, CISCO 2002, IEEE 2004] Message integrity using this key generator relies on matching the authentication data point with a data point in the message header. An initialization vector block precedes the authentication coordinates and any protocol change identifiers in the message. There are no un-used bytes in the authentication block as it is possible to support several different encryption algorithms, each with their own key length. Computation cost using the Synchronized Chaotic System Key Generator is identical for all key sizes. In order to recover the authentication data point in an encrypted message the first ‘n’ blocks must decrypt properly. Failure at this point denies access for the packet contents. Success ensures a strong identity association with the sender.

1.3. Confidentiality Decisions

In the past, the most prevalent choice for ensuring confidentiality of information in the wireless medium was a virtual private network (VPN). [CISCO 2002, Microsoft 2005] CISCO and many other hardware manufacturers allow for configured VPN protection of data in both LAN and WLAN environments. The 802.11i and the 802.1x standards now include implementations of the (Advanced Encryption Standard) AES data and control (access key) encryption on top of any hardware or software generated VPN traffic. Large networks use either the Extensible Authentication Protocol with Transport Layer Security (EAP-TLS) or Protected Extensible Authentication Protocol (PEAP) under the new standards.

The Extensible Authorization Protocol used in conjunction with the Transport Layer Security protocol (EAP-TLS) provides strong authentication with a RADIUS server and is sometimes called Tunneled Transport Layer Security. EAP-TLS is the open source version of transport layer security mechanisms. TLS is also sometimes referred to as the successor to Secure Socket Layer protocol (SSL). [CISCO 2002, Microsoft 2005] Not to be out done, CISCO Systems teamed up with Microsoft to develop their own version called PEAP. It is simply called protected EAP, which uses a protected Message Integrity Check or MIC to authenticate the communications.

Similar to EAP and its variations, messages sent using the Synchronized Chaotic System Key Generator described in this document as the source of encryption keys also contain encrypted data. The largest difference is key exchange. Keys do not need to be exchanged. Depending on the level of security, keys may computed at both ends and change with every block in a message, or at random intervals in a conversation between hosts. The only requirement is that the key rotation scheme is agreed upon between all intended hosts at the beginning of any secure communication. The mechanism that ensures integrity also ensures confidentiality.

1.4. Mobile Concerns

[Eronen 2006, Nordmark 2005, Nikander 2005] Mobile IPv4 and IPv6 may make networks considerably less secure. The security problems do not arise out of the protocols as much as they sprout from five different classes of network configurations that mobile traffic may cross at any point between peers. The five security classes are Public Content Servers, Open Configuration, Role Based Access Control, Cryptographic Security, and Loose Association between peers. It is the last point (on association between peers) that the IETF Requests For Comments (RFCs) do not fully agree. One working group may classify an association as loose, while another identifies the same association as strong.

This discussion does not address this issue in detail, but it does allow both hosts in a conversation to contact a central home service or system to re-establish a secure mobile conversation when one or more of the hosts change IP address or move to a different network. Once the peer to peer connection is re-established, encryption using this Synchronized Chaotic System Key Generator can resume as if the connection had never been broken or intermediary routes had not changed.

1.5. Transport Layer Security

[Dierke 2006] The Transport Layer combines two different protocol layers into an efficient means for securing network communications while ensuring end to end connectivity between hosts. The highest layer in the Transport Layer Security (TLS) protocol is the TLS handshake. It consists of a Diffie-Hellman (DH) key agreement based on either the Digital Signature (DSA) or the Rivest Shamir (RSA) encryption algorithms. The DH key is then used to secure the lower layer of TLS, the TLS Record Protocol.

[Dierke 2006] TLS is used when the communication between peers is private. Symmetric or private/public key pairs may be used depending on the implementation. All packets contain a message integrity check field and are encapsulated using this protocol. Applications and higher layer protocols must pass their data through this layer before their information can be sent or received. The transport layer is the ideal location to encrypt data using the Synchronized Chaotic System Key Generator because it is the lowest layer at which encryption may be applied, and all incoming and outgoing communication between hosts must pass through this layer.

1.6. Constrained Systems

Many mobile clients may also be constrained in memory space and in the amount of encrypted data traffic that can supported by their available bandwidth. [Blake-Wilson 2006] One way around this problem may become the next standard for all IP accessibility, authentication, and accounting control in the Transport Layer. Transport Layer Security is the ideal location to use the Synchronized Chaotic Systems Key Generator. Elliptic Curve Cryptography (ECC) is implemented using the Diffie-Hellman key exchange protocol, and is not susceptible to eavesdropping by outsiders. The Diffie Hellman protocol establishes a strong connection between all hosts and the authentication server or any other authentication, authorization and accounting protocol in use to restrict resource access. The protocol provides forward security when the client requests the server use ephemeral keys (ECDHE). Forward security is not possible when using a fixed key (ECDH), or in anonymous authentication which provides little security and is open to eavesdroppers.

Currently, the digital signature algorithm (ECDHE_ECDSA) or the Rivest-Shamir algorithm (ECDHE_ ECRSA) protocols are used with signed ephemeral keys to provide forward security. The server and the mobile host can verify that information has been sent from an authorized entity by examining information transmitted within the message. If there is a disagreement and data cannot be verified as originating from an authorized sender, the packets are dropped.

Small ECC keys can provide higher security than much larger DH/DSA/RSA encryption keys because they can change with every packet encrypted. [Blake-Wilson 2006] The largest ECC key, at 571 bits is equivalent to a 15,360 bit DH/DSA/RSA key. The reduced key size translates into power savings, reduction in hardware requirements, and overall size of the mobile device. The Elliptic Curve Encryption Protocol Suite closely follows the DH/DSA/RSA algorithm protocols. It is possible to select weak keys with ECC. Curves that produce weak keys are circular or consistent, minimally eccentric curves.

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Figure 1.1 Comparable Key Sizes (in bits) [Blake-Wilson 2006]

Weak keys should be avoided. In the case of elliptic curves, circular or nearly circular, and regularly repeating curves produce weak keys because the next step can be more easily guessed than a more chaotic ellipse. This key generator is based on a Flash animation of the Euler Three Body problem by David Harrison, which use derivatives of Newton's equations of motion [Harrison 2005]. Modifications to the implementation avoid planetary orbital patterns that are easily solved using the brute force, known text, and baby step/giant step attacks [Musson 2006]. Newton's equations of motion are analytically solvable, but require the use of elliptic integrals [Musson 2006].

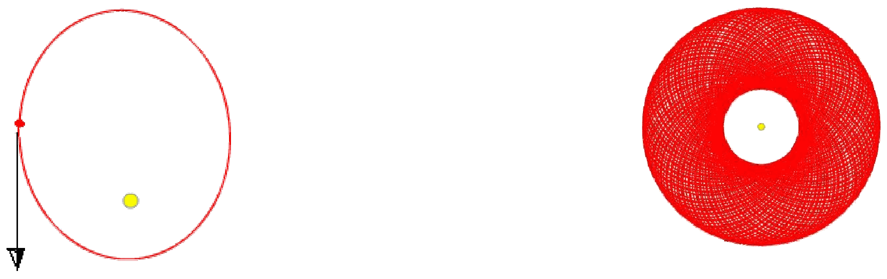


Figure 1.2 Examples of orbits that produce weak keys.

The keys generated from the orbits above are more easily guessed than dynamically changing values because of cyclic repetitions of generator values. [Musson 2006] Forward security of information requires that generators produce chaotic orbits instead of predictable ones because security is increased when dynamic orbits are used to produce encryption keys.

	y_0, vy_0	y_1, vy_1	y_2, vy_2	y_3, vy_3	y_4, vy_4	y_5, vy_5	y_6, vy_6	y_7, vy_7
x_0, vx_0	X							
x_1, vx_1		X						
x_2, vx_2			X					
x_3, vx_3				X				
x_4, vx_4					X			
x_5, vx_5						X		
x_6, vx_6							X	
x_7, vx_7								X

Figure 1.3 Identity OF A Stable Orbit

To capture the generating equation for the examples above using the "Baby Step/Giant Step" attack, less than half of the key values must be correctly guessed. [Musson 2006] The remainder can then be computed as the complement of its orbital position due to the stability of the orbit that the generator is producing encryption keys for. Once the identity of the orbit is revealed in a table similar to Figure 1.3, security is broken because the subset of valid keys is revealed. [Musson 2006] Brute force attacks on the identified subset may then be completed very quickly to determine the current position and interval of the orbit that the key generator is following and thus reveal all encrypted data in the transmission stream. When a dynamic curve is used, each x and y position must be correctly guessed along with the velocity along the x and y axis for any given key. The size of the subset of valid keys is then roughly equivalent to the total set of possible keys.

2. NARRATIVE

For the purposes of this research, no distinction is made between small household wireless networks, large corporate wireless installations, and the mobile telephone industry. The author's research has revealed a convergence in technologies in both the cellular telephone industry and services available to internet users. The Synchronized Chaotic System Key Generator does not address multi-homed solutions, but it may be included as a key generator in peer to peer (P2P) and other wireless and networked solutions where endpoints of an encrypted "conversation" may be composed of any combination of mobile hosts and wired or fixed hosts.

The Synchronized Chaotic System Key Generator generates points along the chaotic path of a planet that is orbiting two fixed stars. The model of the orbit is maintained in double space, between -2 and +2 on the x and y axis respectively. When an output is required for use as an encryption key, the output is scaled to produce only positive unsigned integers. All Cryptographic Block Chaining (CBC) Mode capable, symmetric key encryption algorithms, with total key lengths of 80, 112, 128, 192, and 256 bits long are supported within this key generator implementation.

All operations performed by this key generator are transparent to the user. There are no entry screens. Applications should be able to replace existing key generators for symmetric secure socket protocols with the Synchronized Chaotic System Key Generator for increased speed and information security.

2.1. Transport Layer Encryption

One of several possibilities where this key generator exhibits increased speed and forward security while reducing computational and storage requirements is encryption at the

Transport Layer of the Internet Protocol (IP) stack. The following paragraphs describe a modified four way transport layer security (TLS) handshake using this key generator.

At the beginning of a secure connection or conversation using a protocol based on the Synchronized Chaotic System Key Generator, hosts know the value of gravity for orbital mass (G_s), the starting point (S_1), the value of gravity of one of two large gravity wells (G_{P2}), and the fixed position of both gravity wells [Harrison 2005]. An initial handshake establishes a common dynamic orbit for all hosts in the conversation using the Diffie Hellman protocol to determine the mass of the second large gravity well (G_{P1}). G_{P1} is stored as a floating point value larger than 0.4 and less than 2.0 times the mass of G_{P2}. Once the chaotic orbit is established by determining the mass of G_{P1}, hosts may update their elliptical models only when messages are sent or received in the conversation. This restriction reduces the amount of system resources required to update key generators. Model synchronization is maintained within each message or file transfer by a 2 byte time field which describes which step is used to create the authentication check which immediately follows the encrypted initialization vector at the beginning of the data transfer. The authentication check consists of the Cartesian coordinates of the orbital mass. If the correct coordinates are not identified, the remaining parts of the message are discarded.

The Diffie-Hellman protocol exchange begins with RSA or DSA depending on the private-key signed certificate of the most constrained host. It is assumed that a centralized server is always initially used as an authenticating authority. The server has one function in the initial exchange, to send private key signed values of ($G_{(s1,\ s2)}$) to each of the hosts in the new conversation. Each host then derives $G_{(s1,\ s2)}$ of each message for the other hosts and multiplies each $G_{(s1,\ s2)}$ to form gravity G_{P1} of the elliptical model. The Diffie-Hellman protocol is then used to arrive at a common, encryption/decryption algorithm "in use" value. AES, DES, 3DES symmetric encryption algorithms is available within the protocol extension. The hosts exchange each other's IP addresses using the common encryption/decryption algorithm in the protocol to complete the handshake sequence. Error recovery at this point re-initializes all values, and the process must be repeated.

The time component transmitted as part of the message header synchronizes the model between conversation participants. Each time a message is received, the time value is used to compute the initial key for decryption based on the value of the protocol change data contained in the authentication block. This protocol change data contained in the first byte of the authentication block determines the length of the key to be generated and the encryption algorithm that was used to generate the message. There are five algorithms supported by this key generator.

The following modified 4-way handshake must be observed before any application data can be transmitted. This is similar to the 4-Way handshake found in IETF RFC 4492 [Blake-Wilson 2006] with a final model verification check which concludes the exchange before application data begins to flow from the client to the server. Please refer to Appendix A (Data Definitions) for definitions of each item in the modified exchange below. Appendix B contains a diagram of the entire 4-way handshake including the name of each message sent, or action by the respective host.

The client initiates the handshake by sending a hello message with an algorithm request consisting of a 16 byte block signed with the client's public encryption key. The server responds with its own hello message. It then sends change cipher spec, its own certificate, DSA or RSA public key, certificate request to the client, and algorithm support response messages to the client.

The server's certificate must contain an either an ECDSA or an ECRSA capable public key and signed with either ECDSA or ECRSA respectively to provide forward secrecy. The client generates an ECDHE key based on the server's public key and along the same curve as the server's ephemeral ECDHE key and sends its own certificate using the server's ECDSA/ECRSA public key in the ClientKeyExchange message. The client also verifies the server's certificate with a certificate authority and sends the server a request for the mass of G_{P1} and which encryption algorithm it prefers to use.

The server verifies the client's certificate, then sends a value for the mass of G_{P1} and echoes the value of the encryption algorithm. It then changes to the agreed-upon data transfer cipher specification and sends the client's IP address. The client, upon decrypting its own IP address will send a done message to the server using the data transfer cipher specification. Regular data transfer can now begin using the protocol.

2.2 Data Flow Following Initial Handshake

Each regular data message consists of the following segments. 1) The number of key computation cycles before output of the check value components. 2) The encrypted message beginning with an initialization vector consisting of sixteen random characters followed by an encrypted random skip value together with the Cartesian check coordinates. If the check coordinates are correct, the remaining blocks of the message are decrypted using the new skip value as the number of computation cycles to complete before a key is ready for the encryption/decryption algorithm. The skip value reduces the probability of discovery of the elliptic curve using the "Baby/Giant Step Attack". The attack is an exhaustive or brute force known text attack where many curves are created with tiny elliptic integrals used to discover the key generating ellipse. Refer to Appendix B for an example of this modified 4-way TLS handshake. [Musson 2006]

3. KEY GENERATOR DESIGN

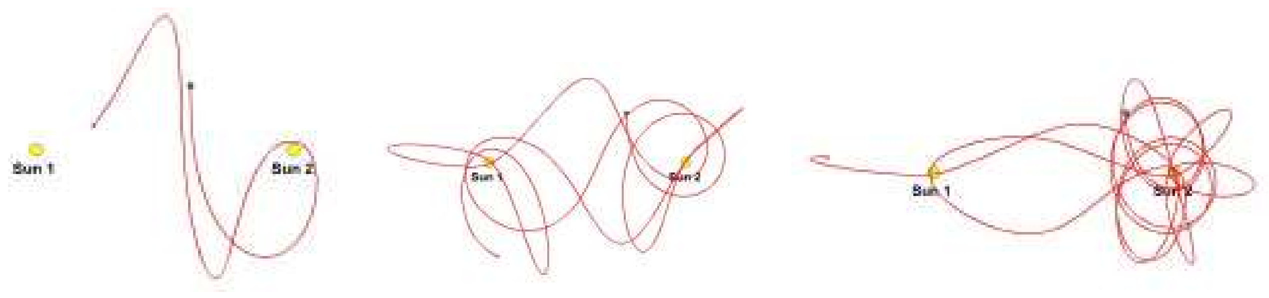
3.1. Chaotic System Selection

This key generator is the direct marriage of standard encryption key generation algorithms and astronomical orbits of stars around black holes or planets in orbit around stars. [Harrison 2005] The Euler 3 body problem provides higher security since orbits of objects are more complex. An object of fixed mass in orbit between two other objects, also of, fixed mass can be identified by five components. The first two components are the (x, y) coordinate value of the object's position. The third and fourth components are the acceleration vector of the object expressed as the velocity along the x and y axes. Finally, the fifth component is the time step of the position sample. These values are set in a specific order and scaled to produce a series of encryption keys which can be used to encrypt/decrypt data on a block by block basis within a packet, or an entire file transfer regardless of the number of sixteen byte blocks. The key generation model and scaling algorithms run in real-time and cost less in computation time than the encryption/decryption of data.

3.2. Synchronization

Asymmetry in chaotic models stem from two basic sources. The first source of asymmetry is variances in the model itself. The second variance stems from how the system running the model interprets the generated information. The next paragraph illustrates variances in chaotic models as a result of changing one of many variables used to create the model.

The images in figure 3.1 were generated using a 'Flash animation' written by David M Harrison at the University of Toronto. [Harrison 2005] The only variable that was changed to generate the curves was the initial position along the x axis. [Harrison 2005] The mass of star 1 is 1.5 times the mass of star 2 in each of these examples. Dynamic curves are plotted using a red dot every 15 iterations of the generator loop. An encryption key is taken from the generator every time a point is determined. It consists of the following five values in this order: x, y coordinates, the velocity along the x and y axes, and the total number of times the generator cycle has been called. Each value is represented by an unsigned integer.



Small Initial x-position Intermediate Initial x-position Large Initial x-position

Figure 3.1 Variances in Model Constants

Through experimentation and analysis of resulting orbital patterns several limitations were required to force the model to always run inside a finite field. When the model exceeded its finite field, the simulation terminated. Each variable and model constant was tested to identify optimum response time and limit boundaries. Response time and model stability depend largely on the derivative of time, or more simply the time slice of the computation. If the value is too large, the model is rough and unstable. Smaller values are preferable, but there is a lower limit as well. When the time slice is too small, the model simulation does not progress fast enough for use as an encryption key generator. Extremely small time slice values are also not desirable because they open the generator to an attack known as the "Baby Step Attack." [Musson 2006]

Variances between computers and operating systems introduce errors into the model as well. The design of the key generator includes two large tables or arrays for standardizing square root calculations, and avoids the use of mathematical “power” functions available in many programming languages. Specialized math functions produce an approximation, which is unacceptable for cryptography since all hosts must arrive at an identical position in the dynamic orbit of the key generating model to produce identical encryption keys. The key generator is stabilized by a paired lookup tables that eliminate host variances in the computation of velocity along the x and y axes.

4. TESTING AND EVALUATION

4.1. Code Analysis

The Synchronized Chaotic System Encryption Key Generator was designed to have a small memory footprint and operate in real time. There are no global variables, and all named variables are declared, initialized, and maintained by the software's main driver. The software uses an integer returning function to search a lookup table for the largest value that is less than or equal to the function input value. The sqrt_id function is the heart of how this chaotic model is always interpreted the same on each system it is run on.

On many systems square roots are calculated as approximations. This means that one system may interpret an answer for the square root of eighteen as 4.242640687 while another may interpret the value to be 4.242640686 or 4.242640688. The implementation gets around this limitation by searching for the largest value that is less than or equal to the input value, then inserts a pre-computed value for the ceiling of the square root into the appropriate equation. Equation elements that are a value which is squared or cubed are simply multiplied against themselves until the required power is reached.

Structural variables are always passed to functions by reference. Functions that are used to produce encryption keys extract required components from the structures and place the values in local variables before performing computations with the information. Unsigned character arrays are passed along with the length of the array to ensure that the proper amount of data is read from or inserted into the array to prevent buffer underflow and overflow.

4.2. Black Box Testing

Black box testing was performed on the operational code by executing it on various systems with very different capabilities. The key generator has proven that all hosts capable of performing multiplication and addition of the floating point atomic data type “double” is capable of producing identical keys given an identical key generator value for the gravitational mass of the variable star or gravity well. The orbital mass always accelerates towards one of the fixed stars. Apparent deceleration or slowing of an orbital mass is also a direct affect of a star’s gravitational pull. This key generator calculates and applies the acceleration of the orbital mass for each position. The square root of a value is part of the acceleration equation. Experience has shown that individual hosts may compute the square root as a slightly different value. Therefore, a paired stabilization table was developed to ensure that identical answers are for all square root calculations to be performed by the key generator. This key generator may be used on hosts which are able to add, multiply, and divide thirty two bit numbers.

During the black-box test phase, a fault in how the encryption key is recovered from the storage array was discovered. This flaw created what appeared to be a variable number of bytes in the output of all keys. The problem was most noticeable in larger keys. The solution to this output problem was to correct a bug in the conversion from binary input to hexadecimal output.

Appendix C is a run-time analysis of the key generator. In order to simulate a highly constrained host a large amount of operating system messaging and other overhead was placed on the test machine by using a Perl script to call the key generator program that was written in the C programming language. The cost is measured as the time it takes to generate a key and perform a hexadecimal conversion. An arbitrarily large number (77,777) was chosen for the maximum keys to generate along a dynamic orbit.

Each data set ran for approximately thirty six hours on a dedicated test machine, an IBM R50e laptop with a 1.50GHz Intel® Pentium® M processor, 239MHz front-side bus and 1.24 GB ram, running FreeBSD Linux version 6.2. Each data set was then plotted using Microsoft

Excel and average cost computed. It was interesting to note that the cost of an eighty bit key including all of the overhead was equivalent to 98% of the cost of a 256 bit encryption key. A combination of the operating system overhead incurred by using a Perl script to call the key generator along with the performance of the optimal binary searches performed within the generator are visible in the form of S shaped patterns in the outliers in the plotted data.

4.3. Attack Resiliency

Attacks against elliptical curve generated keys are similar to those employed to break keys generated by any other means. [Musson 2006] Typically attacks fall into two categories, passive and active attacks. The known text attack is an active attack where the attacker tries to generate the key from a series of messages. The major difference between this protocol and all other protocols is that each message requires a random set of keys for decryption. Active and passive attacks will require a very large sample of encrypted data to guess decryption key sets. Each key output from the key generator used to encrypt data in cryptographic block chaining mode (CBC) for one block only exponentially decreases the chance that any proper key could be guessed. Encryption protocols that utilize key transmission protocols may be reduced to sending the time slice value or the number of steps to skip to compute any given key before using that key. Remote hosts in the conversation compute the new key given in the key transmission message. Actual symmetric keys no longer need to be transmitted when a pre-shared key protocol is used, increasing security by reducing the attack surface.

4.4. Generator Output

```
---------------------------------------------------------
orbital planet is at:
  time =  1
   x = 0.20116522530723716
   y = -0.49645106964041874
  vx = 1.16522530723714146
  vy = 3.54893035958124736
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  80 bit key  = CBE072EABCFF59063663
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  112 bit key = 5906CBE0F8DE3663BCFF72EA58EA
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  128 bit key = 36631361CBE0F8DE72EA58EA5906BCFF
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  192 bit key = 59066F046F04CBE0F8DEF8DE72EA58EA58EABCFF36631361
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  256 bit key =
72EA4CEC1AE0F8DEB9E0C1E01FEC8FEBEAFFF00536631361FEFFF96180056F04
---------------------------------------------------------
---------------------------------------------------------
orbital planet is at:
  time =  153
   x = 0.43622097993037989
   y = 0.00275222396183740
  vx = -16.29121268449059201
  vy = -35.74176859001322271
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  80 bit key  = F3E910D9FFFF48503144
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  112 bit key = 4850F3E91AE83144FFFF10D942D7
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  128 bit key = 31445949F3E91AE810D942D74850FFFF
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  192 bit key = 485048504850F3E91AE81AE810D942D71CD7FFFF31445949
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  256 bit key =
10D968D839E91AE8CCE9E9E910D942D762F948505949D44266FF31449D533B4D
---------------------------------------------------------
```

Figure 4.1: Generator Output With Source Information

This example shows that sample points taken from the key generator produce very different encryptions keys. Each two byte field in the keys is the result of scaling operations using prime numbers for scaling calculations. The key generator uses all components for tracking the position and velocity of the orbital mass as inputs to scaling functions which convert the generated data into each hexadecimal key.

4.5. Key Generation Rates On Constrained hosts

Each key generating cycle requires an average of 6.823 microseconds. Writing 65,535 iterations to file for each orbital position and encryption key size took quite a bit longer than computing that many keys and was limited to the access speed of the storage device of the test system.

Real power savings will be realized by constrained hosts via the absence or reduction of key transmission protocols for encryption/decryption keys along with smaller computational loading for key generation. Wireless radio transmissions reduce battery life because they require more energy than computation cycles. Bandwidth that might be better utilized for data is also reduced whenever keys are transmitted over the wireless network.

5. RESULTS AND CONCLUSION

5.1. Energy Savings With Improved Security

Current public/private key encryption algorithms require transmission of the new public key each time a private key is generated along an elliptic curve. This restriction reduces the number of key rotations that are possible within any given length of time. The most secure key transmission protocols send one (encrypted) bit at a time until the entire key is successfully decrypted by the receiving host. That is 160 messages per key change including all of the response messages from the receiving host. Then there is the computational time consumption by both hosts when sending the key exchange messages. Data transmission is blocked until the new key is received and verified. The elegance of elliptical and chaotic system key generators is that keys can be rotated after each data message, making the data stream more secure, but when the total of the number of key messages sent is compared to the number of data messages, the data rate may fall as low as 6.25% of all messages in the data stream. Symmetric key cryptography using elliptic curve key generators are also required to transmit encryption keys using the key transmission protocol. The Synchronized Chaotic System Key Generator reduces key transmission requirements for symmetric key cryptography to a subscript value which is ultimately computed by all intended hosts. The key generator also supports encryption protocols that require extremely high key rotation rates.

5.2. Host Workload Measurements

Several applications were created to test the algorithm. They were coded in the C programming language and executed on various systems with different capabilities. Time measurements were taken for each system. Best-case completion times varied between 100 and 300 milliseconds with an average completion time of 150 milliseconds to compute 65,430 key generator values. Average-case computation times varied between 1.6 microseconds and 2.79 seconds with an average computation time of 32.19 microseconds. The average cost of an 80 bit encryption key is equivalent to 98% of the cost of a 256 bit encryption key. Refer to Appendix C for performance charts and numerical analysis of each encryption key size.

Elliptic Curve Cryptography is most attractive because of the reduction in computational cost for encryption keys. [NSA Case] The worst case cost comparison for ECC keys is three times less expensive, to a best case of 64 times less expensive to generate than comparative strength Diffie-Hellman keys. The NSA requires large base field primes for use in computing the elliptical curves. This key generator produces encryption keys of comparable strength in a fraction of the time required by NSA algorithms because it takes a novel approach to generating encryption key sequences.

5.3. Conclusion

The state of wireless connectivity is in constant flux, always expanding. Security requirements will therefore remain in a constant state of review. Today's most secure choices by review and/or testing organizations like the IEEE, IETF, and the Wi-Fi Alliance may be broken by an obscure flaw in one protocol or encryption algorithm. The future for new protocols based on elliptic curve cryptography based on ephemeral keys looks very promising. Solid encryption algorithms like AES, DES, 3DES, DSA, and RSA can be improved by how data and keys are passed through the existing algorithms. History has proven that key rotation is critical to forward secrecy of communications over any network. The Synchronized Chaotic System Symmetric Key Generator does not seek to become a magic bullet to solve all security problems. The goal in design of this key generator is to provide another secure option for application developers and further thwart unethical and illegal compromise of private data transmitted over non-secured networks.

BIBLIOGRAPHY AND REFERENCES

[Ameter 2003] Christopher R. Ameter, Russell A. Griffith, John K. Pickett, Chris Prosise: "WHIFF – Wireless Intrusion Detection System". Foundstone®, Inc. 2003

[Arabasz 2002] Paul Arabasz. Judith Pirani. "Wireless Networking at Carnegie Mellon University: ECAR Case Study 6" EDUCAUSE. Boulder, CO: 2002

[Blake-Wilson 2006] S. Blake-Wilson et al. "RFC-4492: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)." The Internet Society. May 2006

[Chou 2005] R. Chou et al. "RADIUS Vulnerabilities in Wireless and Wired Environments." IETF Networking Group. July 28, 2004 <http://www.drizzle.com/~aboba/RADEXT/radius_vuln_00.txt> Accessed June 2005

[CISCO 2002] "A Comprehensive Review of 802.11 Wireless LAN Security and the Cisco Wireless Security Suite." Cisco Systems, Inc. 2002 http://www.cisco.com/en/US/netsol/ns339/ns395/ns176/ns178/networking_solutions_white_paper09186a00800b469f.shtml Accessed June 2006

[Dierke 2006] T. Dierke, E. Rescorla. "RFC-4346: Transport Layer Security (TLS) version 1.1." The Internet Society. April 2006

[IEEE 2004] IEEE Computer Society. "IEEE Std 802.11i™ Part 11, Amendment 6." The Institute of Electrical and Electronics Engineers, Inc. New York, NY. 2004

[Eronen 2006] Ed P. Eronen. "RFC4555: IKEv2 Mobility and Multihoming Protocol (MOBIKE)." Nokia, June 2006

[Huston 2005] Ed G. Huston, Ed I. Leuca. "RFC-3975: OMA-IETF Standardization Collaboration." The Internet Society. January 2005

[Harrison 2005] David M. Harrison, "Flash Animation: 3-Body Gravitational Problem." Toronto, Canada, 2005. http://www.upscale.utoronto.ca/GeneralInterest/Harrison/Flash/Chaos/ThreeBody/ThreeBody.html Accessed October, 2006

[Johnson 2003] Worth Johnson. "Secure Wireless Networking at Simon Fraser University." EDUCAUSE Quarterly: November, 2003

[Microsoft 2005] "Overview of the WPA wireless security update in Windows XP." Microsoft Corporation, September 27, 2005 http://support.microsoft.com/?kbid=815485 Accessed June 2006

[Musson 2006] Matthew Musson. "Attacking the Elliptic Curve Discrete Logarithm Problem." Acadia University: Wolfville, Nova Scotia. 2006

[NSA Case] National Security Agency. "The Case for Elliptic Curve Cryptography." http://www.nsa.gov/ia/industry/crypto_elliptic_curve.cfm Accessed November 5, 2006

[Nikander 2005] P. Nikander et al. "RFC-4225: Mobile IP Version 6 Route Optimization Security Design Background." The Internet Society. December 2005

[Nordmark 2005] E. Nordmark, T. Li. "RFC-4218: Threats Relating to IPv6 Multihoming Solutions." The Internet Society. October 2005

[O'Hara 2005] Bob O'Hara et al. "RFC 3990: Configuration and Provisioning for Wireless Access Points (CAPWAP) Problem Statement" The Internet Society February 2005.

[Python.org] "Code Snippet: Tests ECC cipher suites using ssltest" Python.org. http://svn.python.org/projects/external/openssl-0.9.8a/demos/ssltest-ecc/ssltest.sh Accessed 9/4/2006

[Riley 2003] Steve Riley, "Wireless security with 802.1x and PEAP." MCS Trustworthy Computing Services. January 13, 2003

[Wagner 1997] David Wagner, Bruce Schneier, John Kelsey. "Cryptanalysis of the Cellular Message Encryption Algorithm." University of California, Berkeley. 1997

[Wi-Fi 2003] "Enterprise Solutions for Wireless LAN Security." Wi-Fi Alliance February 6, 2003 http://main.wi-fi.org/membersonly/getfile.asp?f=Whitepaper_Wi-Fi_Enterprise2-6-03.pdf Accessed June 2006

APPENDIX A: DATA DEFINITIONS

A.1. VARIABLES

orbital planet:	Stores the current location of the orbital mass Used as the scaling function input for encryption key output
time:	Current time-step or iteration of the model (initialized to a value of 1)
x:	Current position along the x-axis (initialized to a value of 0.2)
y:	Current position along the y-axis (initialized to a value of -0.5)
vx:	Current velocity along the x-axis (initialized to a value of 0)
vy:	Current velocity along the y-axis (initialized to a value of 1)
m:	Mass ratio of Sun 1 vs. Sun 2
rmin:	Minimum radius the star's gravity effects its surrounding space
dt:	The Derivative of time Smaller increments create a smoother, but slower running model. Larger increments create an unstable model that runs very quickly to termination.
skip:	The number of iterations to skip before generating an encryption key output.

A.2. FUNCTIONS

f(x,y)	The derivative of vx with respect to time
g(x,y)	The derivative of vy with respect to time
sqrt_id ()	Provides a ceiling of function for square root calculations by returning the index of the input value from the lookup table. The index in the square root table is identical to the returned unsigned integer value.
calculate_key()	Calculates the current position of the orbital mass.
mk_eighty_bit()	Scales input orbital values, fills related structure, and memcopies the structure into the eighty bit key (unsigned character) array.
mk_one_twelve_bit()	Scales input orbital values, fills related structure, and memcopies the structure into the one hundred twelve bit key (unsigned character) array.

mk_ one_twenty_eight_bit()	Scales input orbital values, fills related structure, and memcopies the structure into the one hundred twenty eight bit key (unsigned character) array.
mk_ one_ninety_two_bit()	Scales input orbital values, fills related structure, and memcopies the structure into the one hundred ninety two bit key (unsigned character) array.
mk_ two_fifty_six_bit()	Scales input orbital values, fills related structure, and memcopies the structure into the two hundred fifty six bit key (unsigned character) array.
dieWithError()	Prints an error message and terminates the test simulation when the generator exceeds its finite field.
RSA	Rivest Shamir Algorithm: This algorithm uses a public and a private key for encryption and document signing.
DSA	Digital Signature Algorithm: The US Federal Government standard for public/private key encryption.

Elliptic Curve Encryption Suite:

ECDH	Elliptic Curve Diffie Hellman encryption protocol
ECDSA	Digital Signature Algorithm implemented using elliptic curves.
ECRSA	Rivest Shamir Algorithm implemented using elliptic curves.
ECDSA_ECDH	
ECRSA_ECDH	
ECDHE	Elliptic Curve Diffie Hellman Ephemeral encryption protocol: Ephemeral use of elliptic curves provide forward security.
ECDSA_ECDHE	Ephemeral Diffie Hellman DSA encryption protocol
ECRSA_ECDHE	Ephemeral Diffie Hellman RSA encryption protocol

APPENDIX B: MODIFIED FOUR WAY TLS HANDSHAKE

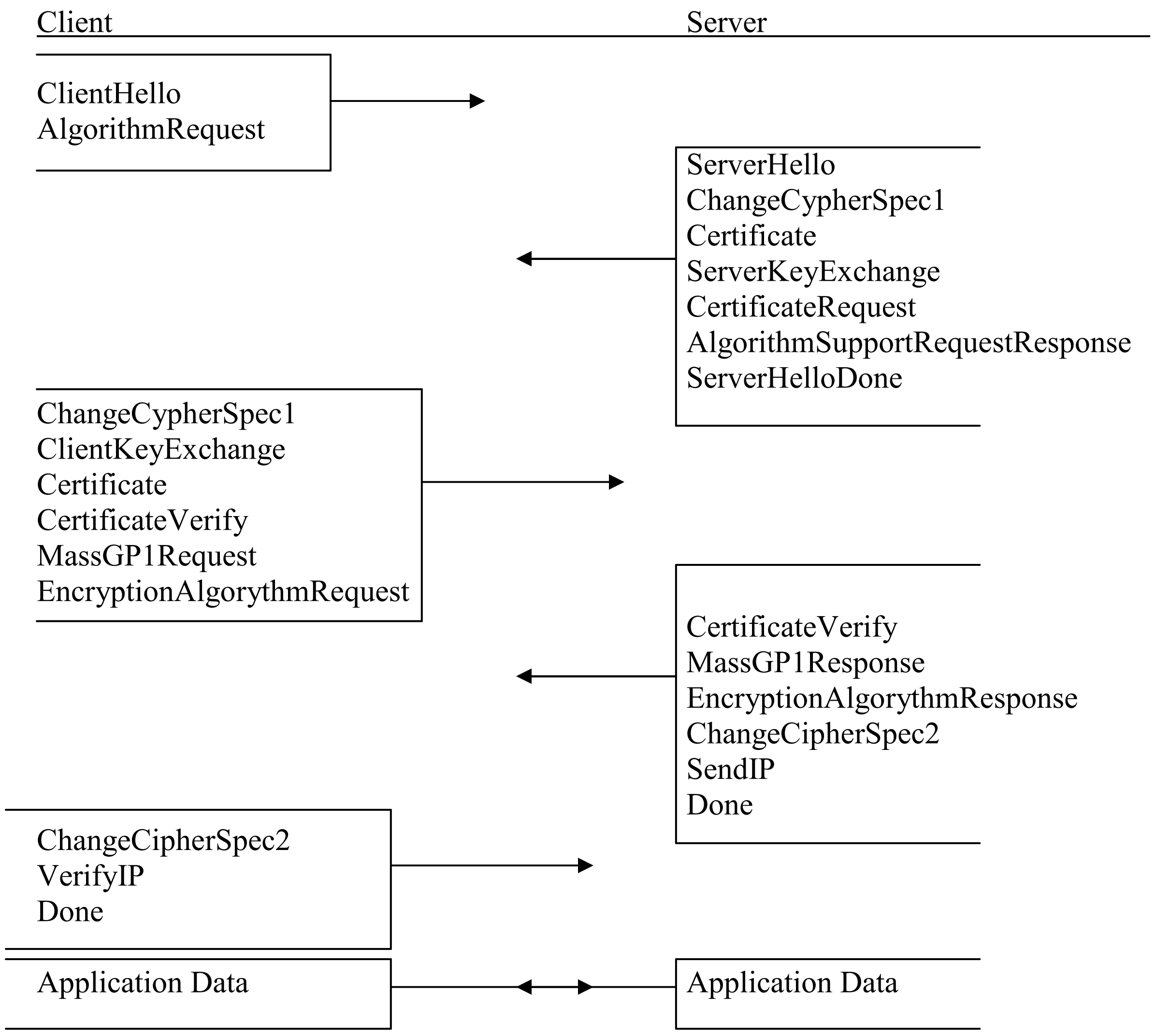
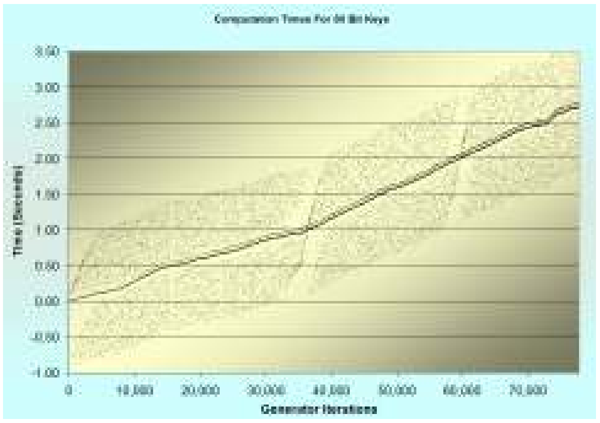
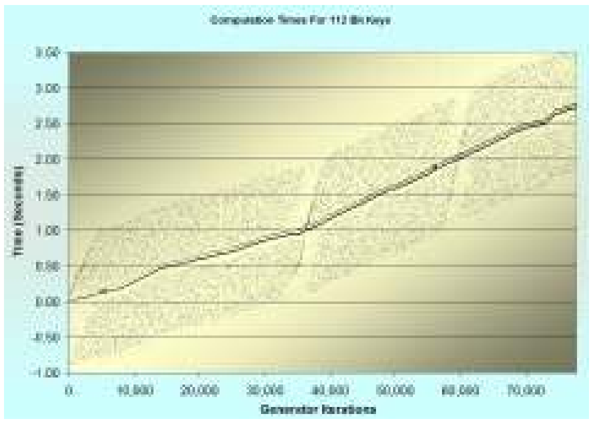


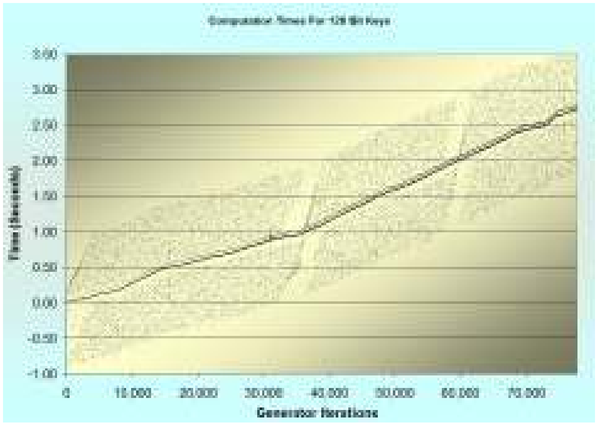
Figure 3.2 Modified 4-Way TLS Handshake

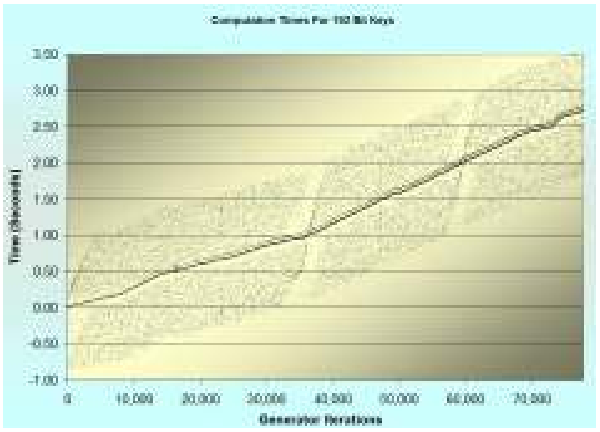
APPENDIX C: AVERAGE CASE KEY GENERATOR RUN TIMES

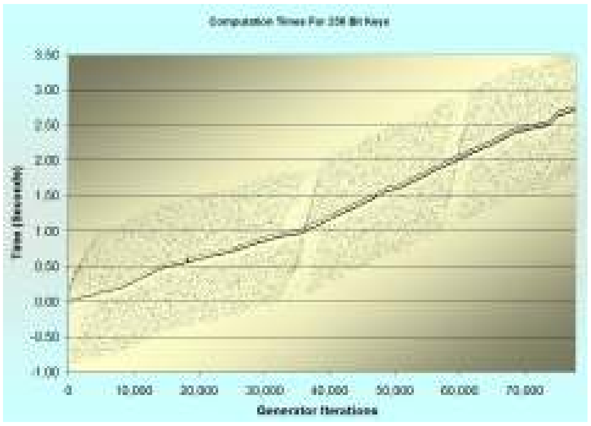
	AVG	COMPARISON					AVG
KEY SIZE	TIME	80	112	128	192	256	Diff
80	31.973 usec	0	-0.2749	-0.0829	-0.1186	-0.6104	-0.217
112	32.248 usec	0.2749	0	0.1921	0.1563	-0.3354	0.0576
128	32.056 usec	0.0829	-0.1921	0	-0.0357	-0.5275	-0.134
192	32.092 usec	0.1186	-0.1563	0.0357	0	-0.4918	-0.099
256	32.583 usec	0.6104	0.3354	0.5275	0.4918	0	0.393











APPENDIX D: SOURCE CODE

** NOTE **   Stabilization tables are not included with this source code.

```
/***************************************************************************
*
*  Cryptographic Encryption Key Generator based on the Euller 3 body problem.
*
*  Adapted from:
*    David M. Harrison, "Flash Animation: 3-Body Gravitational Problem."
*        Toronto, Canada, 2005.
*
*  Adapted By:  Anthony G. Weitekamp
*  Date:        January, 2007
*
*  The key generator produces a sequence of encryption keys along the orbit
*  of the planet.
*
*  program call:  keyGen a b c d
*    Where:
*   a = keySize  Default 5, Range 1 -> 5
*                1 =  80 bit key
*                2 = 112 bit key
*                3 = 128 bit key
*                4 = 192 bit key
*                5 = 256 bit key
*
*   b = m        Default 4000, Range 4000 -> 20000
*   c = skip     Default 3
*   d = maxKeys  Default 0  Indicates the key subscript to be generated
*
*
*   compile using g++ keyGen.cpp -o keyGen
*
***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include "lookup_table.h"  // Stabilization Tables

#ifndef uint8
#define uint8 u_char
#endif
#ifndef uint16
#define uint16 u_short
#endif
#ifndef uint32
#define uint32 u_int
#endif
#ifndef uint64
#define uint64 u_long
#endif
```

```
typedef struct {
   unsigned short    time;  //  initial value   =    1
   double               x;  //  initial value   =  0.2
   double               y;  //  initial value   = -0.5
   double              vx;  //  initial value   =    0
   double              vy;  //  initial value   =    1
} orbital;

typedef struct {
   uint16       x;    // 16 bits
   uint16       y;    // 16 bits
   uint16    time;    // 16 bits
   uint16      vx;    // 16 bits
   uint16      vy;    // 16 bits
              // total   80
} eighty_Bit_key;

typedef struct {
   uint16      vx;    // 16 bits
   uint16       x;    // 16 bits
   uint16      x_;    // 16 bits
   uint16      vy;    // 16 bits
   uint16    time;    // 16 bits
   uint16       y;    // 16 bits
   uint16      y_;    // 16 bits
              // total  112
} one_twelve_Bit_key;

typedef struct {
   uint16   vy;    // 16 bits
   uint16  vy_;    // 16 bits
   uint16    x;    // 16 bits
   uint16   x_;    // 16 bits
   uint16    y;    // 16 bits
   uint16   y_;    // 16 bits
   uint16   vx;    // 16 bits
   uint16 time;    // 16 bits
          // total  128
} one_twenty_eight_Bit_key;

typedef struct {
   uint16   vx;    // 16 bits
   uint16  vx_;    // 16 bits
   uint16 vx_1;    // 16 bits
   uint16    x;    // 16 bits
   uint16   x_;    // 16 bits
   uint16  x_1;    // 16 bits
   uint16    y;    // 16 bits
   uint16   y_;    // 16 bits
   uint16  y_1;    // 16 bits
   uint16 time;    // 16 bits
   uint16   vy;    // 16 bits
   uint16  vy_;    // 16 bits
          // total  192
} one_ninety_two_Bit_key;
```

```
typedef struct {
   uint16     y;    // 16 bits
   uint16    y_;    // 16 bits
   uint16   x_1;    // 16 bits
   uint16   x_2;    // 16 bits

   uint16     x;    // 16 bits
   uint16    x_;    // 16 bits
   uint16   y_1;    // 16 bits
   uint16   y_2;    // 16 bits

   uint16 time_;    // 16 bits
   uint16    vx;    // 16 bits
   uint16   vy_;    // 16 bits
   uint16  vy_1;    // 16 bits

   uint16  time;    // 16 bits
   uint16    vy;    // 16 bits
   uint16   vx_;    // 16 bits
   uint16  vx_1;    // 16 bits
                    // total  256
} two_fifty_six_Bit_key;


//_____________________________BEGIN Prototypes____________________
uint16 sqrt_id (double val);
double f(double x, double y, double rmin, double m);
double g(double x, double y, double rmin, double m);
void calculate_key( orbital &planet, double rmin, double m, double dt);
void mk_eighty_bit    ( orbital &planet, uint8   ecc_80_bit[], int size );
void mk_one_twelve_bit( orbital &planet, uint8  ecc_112_bit[], int size );
void mk_one_twenty_eight_bit(orbital &planet, uint8  ecc_128_bit[], int size);
void mk_one_ninety_two_bit ( orbital &planet, uint8  ecc_192_bit[], int size);
void mk_two_fifty_six_bit ( orbital &planet, uint8  ecc_256_bit[], int size );
int  dieWithError         ( char* errormessage );
//_______________________________END Prototypes____________________


int main (int argc, char** argv) {

    // Declare variables
    uint64  i       =       0;  // loop counter
    uint32  ctr     =       0;  // loop counter
    uint64  count   =       0;
    uint8   keySize =       0;  // loop counter
    uint64  maxKeys =       1;  // Number of keys to generate for testing
    orbital            planet;  // Stores the current location of the orbital
    double                  m;  // Mass ratio of Sun 1 to Sun 2
    double               rmin;  // Minimum radius of both Stars
    double                 dt;  // Derivative of time (or the time slice)
    uint64               skip;  // The number of iterations to skip before
                                // generating an output
    uint8      ecc_80_bit[10];  // ECC Keys as arrays
    uint8     ecc_112_bit[14];
    uint8     ecc_128_bit[16];
    uint8     ecc_192_bit[24];
    uint8     ecc_256_bit[32];
```

```
// Initialize variables
planet.x    =    0.2;
planet.y    =   -0.5;
planet.vx   =      0;
planet.vy   =      1;
planet.time =      0;
dt          =  0.001;
rmin        =  0.065;
i           =      0;

// command-line defaults
keySize     =     80;  // argv [ 1 ]
m           =   4000;  // argv [ 2 ]
skip        =      1;  // argv [ 3 ]
maxKeys     =      0;  // argv [ 4 ]  current value for encryption

if ( atoi ( argv [ 2 ] ) > m && atoi (argv [ 2 ] ) < 200001 )
   m = (double) atoi ( argv [ 2 ] );

if ( atoi ( argv [ 3 ] ) > skip )
   skip = atoi ( argv [ 3 ] );

if ( atoi ( argv [ 4 ] ) > maxKeys )
   maxKeys = atoi ( argv [ 4 ] );

m = (double) m / 10000;


do { // calculate the orbital position
   i++;
   for (ctr = 0; ctr < skip; ctr++) {
      calculate_key( planet, rmin, m, dt );
   }
} while ( i < maxKeys );

// Scale and print the key
if ( atoi (argv [ 1 ] ) == 1 )  {
   memset (ecc_80_bit,  255, sizeof (ecc_80_bit)  );
   mk_eighty_bit ( planet, ecc_80_bit,  sizeof(ecc_80_bit)  );

   for (count = 0; count < 10; count++ ) {
      if (ecc_80_bit[count] < 16) {
         printf("%s", "0");
         printf("%X", ecc_80_bit[count]);
     } else {
         printf("%X", ecc_80_bit[count]);
     }
   }

} else if ( atoi (argv [ 1 ] ) == 2  ) {
   memset (ecc_112_bit, 255, sizeof (ecc_112_bit) );
   mk_one_twelve_bit       ( planet, ecc_112_bit, sizeof(ecc_112_bit) );

   for (count = 0; count < 14; count++ ) {
     if (ecc_112_bit[count] < 16) {
        printf("%s", "0");
        printf("%X", ecc_112_bit[count]);
        } else {
            printf("%X", ecc_112_bit[count]);
        }
       }

    } else if ( atoi (argv [ 1 ] ) == 3  ) {
       memset (ecc_128_bit, 255, sizeof (ecc_128_bit) );
       mk_one_twenty_eight_bit ( planet, ecc_128_bit, sizeof(ecc_128_bit) );

       for (count = 0; count < 16; count++ ) {
         if (ecc_128_bit[count] < 16) {
            printf("%s", "0");
            printf("%X", ecc_128_bit[count]);
        } else {
            printf("%X", ecc_128_bit[count]);
        }
       }

    } else if ( atoi (argv [ 1 ] ) == 4  ) {
       memset (ecc_192_bit, 255, sizeof (ecc_192_bit) );
       mk_one_ninety_two_bit   ( planet, ecc_192_bit, sizeof(ecc_192_bit) );

       for (count = 0; count < 24; count++ ) {
          if (ecc_192_bit[count] < 16) {
             printf("%s", "0");
             printf("%X", ecc_192_bit[count]);
         } else {
             printf("%X", ecc_192_bit[count]);
         }
       }

    } else { //   atoi (argv [ 1 ]) == 5 )
       memset (ecc_256_bit, 255, sizeof (ecc_256_bit) );
       mk_two_fifty_six_bit    ( planet, ecc_256_bit, sizeof(ecc_256_bit) );

       for (count = 0; count < 32; count++ ) {
          if (ecc_256_bit[count] < 16) {
             printf("%s", "0");
             printf("%X", ecc_256_bit[count]);
         } else {
             printf("%X", ecc_256_bit[count]);
         }
       }

    }

    return 0;
}
```

```
// Find the subscript of the largest element that is
// less than or equal to the input value.
// Return the subscript of the largest element that is
// less than or equal to the input value.
uint16 sqrt_id (double val) {

   int i = 0;
   int low, high;

   if ( val < 0 )
      val *= -1;

   low = 0;
   high = 36000;  // maximum filled element in look-up table

   do {

      i = (high - low) / 2;
      if ( look_up[ i ] < val )
        low = i;
      else
        high = i;

      if (look_up[ low + 1 ] > val)
         return low;

   } while ( look_up [ high ] > val ) ;

   return high;
}

// f(x,y) = the derivative of vx with respect to time
// The values r1 and r2 are acceleration components
// This function computes the velocity along the x axis
// with respect to time.
double f(double x, double y, double rmin, double m) {

    double  r1, r2, temp;

    temp = ((1 - x) * (1 - x)) + (y * y);
    if (temp < 0)
       temp *= -1;
    r1 = sqrt_val[ sqrt_id( temp )];

    temp = ((1 + x) * (1 + x)) + (y * y);
    if (temp < 0)
       temp *= -1;
    r2 = sqrt_val[ sqrt_id( temp )];
    if (r1 < rmin) {
       r1 = rmin;
    }
    if (r2 < rmin) {
       r2 = rmin;
    }
    temp = -((x-1)/(r1 * r1 * r1) + m * (1+x)/(r2 * r2 * r2));
    return temp;
}
```

```
// g(x,y) = the derivative of vy with respect to time
// The values r1 and r2 are acceleration components
// This function computes the velocity along the y axis
// with respect to time.
double g(double x, double y, double rmin, double m) {

    double r1, r2, temp;

    temp = ((1 - x) * (1 - x)) + (y * y);
    if (temp < 0)
       temp *= -1;
    r1 = sqrt_val[ sqrt_id( temp )];
    temp = ((1 + x) * (1 + x)) + (y * y);
    if (temp < 0)
       temp *= -1;
    r2 = sqrt_val[ sqrt_id( temp )];

    if (r1 < rmin) {
       r1 = rmin;
    }
    if (r2 < rmin) {
       r2 = rmin;
    }
    temp = -y*(1/(r1 * r1 * r1)+ m * 1/(r2 * r2 * r2));
    return temp;
}


// The orbital position MUST be calculated in this order or else
// the model will not work correctly.
//
void calculate_key( orbital &planet, double rmin, double m, double dt) {

    planet.vx   = planet.vx + dt * f(planet.x, planet.y, rmin, m);
    planet.x    = planet.x + planet.vx * dt;
    planet.vy   = planet.vy + dt * g(planet.x, planet.y, rmin, m);
    planet.y    = planet.y + planet.vy * dt;
    planet.time++;

}


void mk_eighty_bit( orbital &planet, uint8 ecc_80_bit[], int size ) {

     eighty_Bit_key        ecc_80;

     // local copies of generator output
     double                x, y, vx, vy;
     uint16                time;
     uint16                scaled_x, scaled_vx, scaled_y, scaled_vy;
     uint8                 tmp = 255;
     uint8                 ctr = 0;
     // ecc80_[] is an array of characters to hold an 80 bit key
     x           =      planet.x;
     y           =      planet.y;
     vx          =     planet.vx;
     vy           =     planet.vy;
     time         =   planet.time;

     memset (ecc_80_bit, tmp, sizeof(ecc_80_bit) );

     time = 65535 - time;
     memcpy( &ecc_80.time, &time, sizeof(ecc_80.time) );

     if (x < 0) {
        x *=  -1;
        scaled_x    =  uint16 ( double ( 55079 + 9949 * x) );
     } else {
        scaled_x    =  uint16 ( double ( 55541 + 9973 * x) );
     }
     if (y < 0) {
        y *=  -1;
        scaled_y    =  uint16 ( double ( 55079 + 9949 * y) );
     } else {
        scaled_y    =  uint16 ( double ( 55541 + 9973 * y) );
     }
     if (vx < 0) {
        vx *=  -1;
        scaled_vx   =  uint16 ( double ( 55079 + 9949 * vx) );
     } else {
        scaled_vx   =  uint16 ( double ( 55541 + 9973 * vx) );
     }
     if (vy < 0) {
        vy *=  -1;
        scaled_vy   =  uint16 ( double ( 55079 + 9949 * vy) );
     } else {
        scaled_vy   =  uint16 ( double ( 55541 + 9973 * vy) );
     }

     memcpy( &ecc_80.x,   &scaled_x,   sizeof(ecc_80.x) );
     memcpy( &ecc_80.y,   &scaled_y,   sizeof(ecc_80.y) );
     memcpy( &ecc_80.vx,  &scaled_vx,   sizeof(ecc_80.vx));
     memcpy( &ecc_80.vy,  &scaled_vy,   sizeof(ecc_80.vy));
     memcpy( ecc_80_bit,  &ecc_80,      size );
}



void mk_one_twelve_bit(  orbital &planet, uint8  ecc_112_bit[], int size )   {

     one_twelve_Bit_key    ecc_112;

     // local copies of generator output
     double x, y, vx, vy;
     uint16 time;
     uint16 scaled_x, scaled_x_, scaled_y, scaled_y_;
     uint16 scaled_vx, scaled_vy;
     memset (ecc_112_bit, 255, sizeof(ecc_112_bit) );
     x            =    planet.x;
     y            =    planet.y;
     vx           =   planet.vx;
     vy           =   planet.vy;
```

```
time          =   planet.time;
time = 65535 - time;
memcpy( &ecc_112.time, &time, sizeof(ecc_112.time) );

// use max (1) decimal position + prime * value

if (x < 0) {
   x *=  -1;
   scaled_x    =  uint16 ( double ( 55079 + 9949 * x) );
   scaled_x_   =  uint16 ( double ( 55541 + 9973 * x) );
} else {
   scaled_x    =  uint16 ( double ( 55541 + 9973 * x) );
   scaled_x_   =  uint16 ( double ( 55079 + 9949 * x) );
}
if (y < 0) {
   y *=  -1;
   scaled_y    =  uint16 ( double ( 55079 + 9949 * y) );
   scaled_y_   =  uint16 ( double ( 55041 + 9973 * y) );
} else {
   scaled_y    =  uint16 ( double ( 55541 + 9973 * y) );
   scaled_y_   =  uint16 ( double ( 55079 + 9949 * y) );
}
if (vx < 0) {
   vx *=  -1;
   scaled_vx   =  uint16 ( double ( 55079 + 9949 * vx) );
} else {
   scaled_vx   =  uint16 ( double ( 55541 + 9973 * vx) );
}
```

```
     if (vy < 0) {
        vy *=  -1;
        scaled_vy   =  uint16 ( double ( 55079 + 9949 * vy) );
     } else {
        scaled_vy   =  uint16 ( double ( 55541 + 9973 * vy) );
     }

     memcpy( &ecc_112.x,   &scaled_x,   sizeof(ecc_112.x)  );
     memcpy( &ecc_112.y,   &scaled_y,   sizeof(ecc_112.y)  );
     memcpy( &ecc_112.x_,  &scaled_x_,  sizeof(ecc_112.x)  );
     memcpy( &ecc_112.y_,  &scaled_y_,  sizeof(ecc_112.y)  );
     memcpy( &ecc_112.vx,  &scaled_vx,  sizeof(ecc_112.vx) );
     memcpy( &ecc_112.vy,  &scaled_vy,  sizeof(ecc_112.vy) );
     memcpy( ecc_112_bit,  &ecc_112,    size);

}


void mk_one_twenty_eight_bit  (  orbital &planet, uint8  ecc_128_bit[], int
    size ) {

     one_twenty_eight_Bit_key   ecc_128;

     // local copies of generator output
     double x, y, vx, vy;
     uint16 scaled_x, scaled_x_,scaled_y, scaled_y_;
     uint16 scaled_vx, scaled_vy, scaled_vy_, time;

     x            =    planet.x;
     y            =    planet.y;
     vx           =   planet.vx;
     vy           =   planet.vy;
     time         =   planet.time;
     time = 65535 - time;
     memcpy( &ecc_128.time, &time, sizeof(ecc_128.time) );

     if (x < 0) {
        x *=  -1;
        scaled_x    =  uint16 ( double ( 55079 + 9949 * x) );
        scaled_x_   =  uint16 ( double ( 55541 + 9973 * x) );
     } else {
        scaled_x    =  uint16 ( double ( 55541 + 9973 * x) );
        scaled_x_   =  uint16 ( double ( 55079 + 9949 * x) );
     }
     if (y < 0) {
        y *=  -1;
        scaled_y    =  uint16 ( double ( 55079 + 9949 * y) );
        scaled_y_   =  uint16 ( double ( 55041 + 9973 * y) );
     } else {
        scaled_y    =  uint16 ( double ( 55541 + 9973 * y) );
        scaled_y_   =  uint16 ( double ( 55079 + 9949 * y) );
     }

     if (vx < 0) {
        vx *=  -1;
        scaled_vx   =  uint16 ( double ( 55079 + 9949 * vx) );
     } else {
        scaled_vx   =  uint16 ( double ( 55541 + 9973 * vx) );
     }

     if (vy < 0) {
        vy *=  -1;
        scaled_vy   =  uint16 ( double ( 55079 + 9949 * vy) );
        scaled_vy_  =  uint16 ( double ( 55541 + 9973 * vy) );
     } else {
        scaled_vy   =  uint16 ( double ( 55541 + 9973 * vy) );
        scaled_vy_  =  uint16 ( double ( 55079 + 9949 * vy) );
     }


     memcpy(  &ecc_128.x,    &scaled_x,   sizeof(ecc_128.x)  );
     memcpy( &ecc_128.x_,   &scaled_x_,   sizeof(ecc_128.x_) );
     memcpy(  &ecc_128.y,    &scaled_y,   sizeof(ecc_128.y)  );
     memcpy( &ecc_128.y_,   &scaled_y_,   sizeof(ecc_128.y_) );
     memcpy( &ecc_128.vx,   &scaled_vx,   sizeof(ecc_128.vx) );
     memcpy( &ecc_128.vy,   &scaled_vy,   sizeof(ecc_128.vy) );
     memcpy( &ecc_128.vy_, &scaled_vy_,   sizeof(ecc_128.vy_));
     memcpy( ecc_128_bit, &ecc_128,   size );
}



void mk_one_ninety_two_bit(  orbital &planet, uint8  ecc_192_bit[], int size){

     one_ninety_two_Bit_key     ecc_192;

     // local copies of generator output
     double x, y, vx, vy;
     uint16 scaled_x,   scaled_x_,  scaled_x_1;
     uint16 scaled_y,   scaled_y_,  scaled_y_1;
     uint16 scaled_vx,  scaled_vx_, scaled_vx_1;
     uint16 scaled_vy,  scaled_vy_;
     uint16 time;
     x           =    planet.x;
     y           =    planet.y;
     vx          =   planet.vx;
     vy          =   planet.vy;
     time        =   planet.time;
     time = 65535 - time;
     memcpy( &ecc_192.time, &time, sizeof(ecc_192.time) );

     if (x < 0) {
        x *=  -1;
        scaled_x    =  uint16 ( double ( 55079 + 9949 * x) );
        scaled_x_   =  uint16 ( double ( 55541 + 9973 * x) );
        scaled_x_1  =  uint16 ( double ( 55541 + 9973 * x) );
     } else {
        scaled_x    =  uint16 ( double ( 55541 + 9973 * x) );
        scaled_x_   =  uint16 ( double ( 55079 + 9949 * x) );
        scaled_x_1  =  uint16 ( double ( 55079 + 9949 * x) );
     }
```

```
    if (y < 0) {
       y *=  -1;
       scaled_y    =  uint16 ( double ( 55079 + 9949 * y) );
       scaled_y_   =  uint16 ( double ( 55041 + 9973 * y) );
       scaled_y_1  =  uint16 ( double ( 55041 + 9973 * y) );
    } else {
       scaled_y    =  uint16 ( double ( 55541 + 9973 * y) );
       scaled_y_   =  uint16 ( double ( 55079 + 9949 * y) );
       scaled_y_1  =  uint16 ( double ( 55041 + 9973 * y) );
    }

    if (vx < 0) {
       vx *=  -1;
       scaled_vx   =  uint16 ( double ( 55079 + 9949 * vx) );
       scaled_vx_  =  uint16 ( double ( 55079 + 9949 * vx) );
       scaled_vx_1 =  uint16 ( double ( 55079 + 9949 * vx) );
    } else {
       scaled_vx   =  uint16 ( double ( 55541 + 9973 * vx) );
       scaled_vx_  =  uint16 ( double ( 55079 + 9949 * vx) );
       scaled_vx_1 =  uint16 ( double ( 55079 + 9949 * vx) );
    }

    if (vy < 0) {
       vy *=  -1;
       scaled_vy   =  uint16 ( double ( 55079 + 9949 * vy) );
       scaled_vy_  =  uint16 ( double ( 55541 + 9973 * vy) );
    } else {
       scaled_vy   =  uint16 ( double ( 55541 + 9973 * vy) );
       scaled_vy_  =  uint16 ( double ( 55079 + 9949 * vy) );
    }


    memcpy( &ecc_192.x,    &scaled_x,    sizeof(ecc_192.x)   );
    memcpy( &ecc_192.x_,   &scaled_x_,   sizeof(ecc_192.x_)  );
    memcpy( &ecc_192.x_1,  &scaled_x_1,  sizeof(ecc_192.x_1) );

    memcpy( &ecc_192.y,    &scaled_y,    sizeof(ecc_192.y)   );
    memcpy( &ecc_192.y_,   &scaled_y_,   sizeof(ecc_192.y_)  );
    memcpy( &ecc_192.y_1,  &scaled_y_1,  sizeof(ecc_192.y_1) );

    memcpy( &ecc_192.vx,   &scaled_vx,   sizeof(ecc_192.vx)  );
    memcpy( &ecc_192.vx_,  &scaled_vx_,  sizeof(ecc_192.vx_) );
    memcpy( &ecc_192.vx_1, &scaled_vx_1, sizeof(ecc_192.vx_1));

    memcpy( &ecc_192.vy,   &scaled_vy,   sizeof(ecc_192.vy)  );
    memcpy( &ecc_192.vy_,  &scaled_vy_,  sizeof(ecc_192.vy_) );

    memcpy( ecc_192_bit,  &ecc_192,     size );
}
```

```
void mk_two_fifty_six_bit(orbital &planet, uint8  ecc_256_bit[], int size ) {

     two_fifty_six_Bit_key      ecc_256;

     // local copies of generator output
     double x, y, vx, vy;
     uint16 time, time_;
     uint16 scaled_x,  scaled_x_, scaled_x_1, scaled_x_2;
     uint16 scaled_y,  scaled_y_, scaled_y_1, scaled_y_2;
     uint16 scaled_vx, scaled_vx_, scaled_vx_1;
     uint16 scaled_vy, scaled_vy_, scaled_vy_1;

     x           =      planet.x;
     y           =      planet.y;
     vx          =     planet.vx;
     vy          =     planet.vy;
     time        =   planet.time;
     time =  65535 - time;
     time_ = time * 11;

     if (x < 0) {
        x *=  -1;
        scaled_x    =  uint16 ( double ( 55079 + 9949 * x) );
        scaled_x_   =  uint16 ( double ( 55541 + 9973 * x) );
        scaled_x_1  =  uint16 ( double ( 55531 + 9883 * x) );
        scaled_x_2  =  uint16 ( double ( 55373 + 9931 * x) );
     } else {
        scaled_x    =  uint16 ( double ( 55541 + 9883 * x) );
        scaled_x_   =  uint16 ( double ( 55531 + 9973 * x) );
        scaled_x_1  =  uint16 ( double ( 55373 + 9931 * x) );
        scaled_x_2  =  uint16 ( double ( 55079 + 9949 * x) );
     }
     if (y < 0) {
        y *=  -1;
        scaled_y    =  uint16 ( double ( 55079 + 9949 * y) );
        scaled_y_   =  uint16 ( double ( 55541 + 9973 * y) );
        scaled_y_1  =  uint16 ( double ( 55541 + 9883 * y) );
        scaled_y_2  =  uint16 ( double ( 55373 + 9931 * y) );
     } else {
        scaled_y    =  uint16 ( double ( 55541 + 9883 * y) );
        scaled_y_   =  uint16 ( double ( 55373 + 9931 * y) );
        scaled_y_1  =  uint16 ( double ( 55541 + 9973 * y) );
        scaled_y_2  =  uint16 ( double ( 55079 + 9949 * y) );
     }

     if (vx < 0) {
        vx *=  -1;
        scaled_vx   =  uint16 ( double ( 55079 + 9949 * vx) );
        scaled_vx_  =  uint16 ( double ( 55541 + 9973 * vx) );
        scaled_vx_1 =  uint16 ( double ( 55373 + 9883 * vx) );
     } else {
        scaled_vx   =  uint16 ( double ( 55541 + 9883 * vx) );
        scaled_vx_  =  uint16 ( double ( 55373 + 9931 * vx) );
        scaled_vx_1 =  uint16 ( double ( 55079 + 9949 * vx) );
     }
     if (vy < 0) {
        vy *=  -1;
        scaled_vy   =  uint16 ( double ( 55079 + 9949 * vy) );
        scaled_vy_  =  uint16 ( double ( 55541 + 9973 * vy) );
        scaled_vy_1 =  uint16 ( double ( 55373 + 9931 * vy) );
     } else {
        scaled_vy   =  uint16 ( double ( 55373 + 9931 * vy) );
        scaled_vy_  =  uint16 ( double ( 55541 + 9973 * vy) );
        scaled_vy_1 =  uint16 ( double ( 55079 + 9949 * vy) );
     }

     memcpy( &ecc_256.time,  &time,        sizeof(ecc_256.time)  );
     memcpy( &ecc_256.time_, &time_,       sizeof(ecc_256.time_) );

     memcpy( &ecc_256.x,     &scaled_x,    sizeof(ecc_256.x)     );
     memcpy( &ecc_256.x_,    &scaled_x_,   sizeof(ecc_256.x_)    );
     memcpy( &ecc_256.x_1,   &scaled_x_1,  sizeof(ecc_256.x_1)   );
     memcpy( &ecc_256.x_2,   &scaled_x_2,  sizeof(ecc_256.x_2)   );

     memcpy( &ecc_256.y,     &scaled_y,    sizeof(ecc_256.y)     );
     memcpy( &ecc_256.y_,    &scaled_y_,   sizeof(ecc_256.y_)    );
     memcpy( &ecc_256.y_1,   &scaled_y_1,  sizeof(ecc_256.y_1)   );
     memcpy( &ecc_256.y_2,   &scaled_y_2,  sizeof(ecc_256.y_2)   );

     memcpy( &ecc_256.vx,    &scaled_vx,   sizeof(ecc_256.vx)    );
     memcpy( &ecc_256.vx_,   &scaled_vx_,  sizeof(ecc_256.vx_)   );
     memcpy( &ecc_256.vx_1,  &scaled_vx_1, sizeof(ecc_256.vx_1)  );

     memcpy( &ecc_256.vy,    &scaled_vy,   sizeof(ecc_256.vy)    );
     memcpy( &ecc_256.vy_,   &scaled_vy_,  sizeof(ecc_256.vy_)   );
     memcpy( &ecc_256.vy_1,  &scaled_vy_1, sizeof(ecc_256.vy_1)  );

     memcpy( ecc_256_bit,  &ecc_256,    size );
}



int dieWithError ( char* errormessage ) {

    perror (errormessage);
    return EXIT_FAILURE;
}
```