Runtime Monitoring of Malicious Code in a Network

Ibironke A. Ajayi
Federal College of Education, Osiele, Nigeria

tayoajayi@unaab.edu.ng

Olutayo B. Ajayi and O. J. Adeniran
University of Agriculture, Abeokuta, Nigeria

olutayoajayi@gmail.com; adeniranoj@unaab.edu.ng

Abstract

The eXecute Only Memory (XOM) used for protecting systems from unauthorized foreign codes does not completely protect workstations against attackers tampering with the memory using code encryption and integrity verification methods. To address this problem, an architectural technique called Runtime Monitoring of Malicious Code in a Network (RUNMAC) was designed to detect program flow anomalies associated with such malicious codes. This was achieved by verifying program code at the hash block (similar to basic block) level by pre-computing the hash functions that generates Hashed Message Authentication Codes (HMAC) for each hash block verifiable during program execution in memory. To achieve protection against automated attack tools, Elliptic Curve (EC) based Multi-signcryption was used to generate the 128-bit HMAC needed to protect network workstations against memory replay attacks. Furthermore, a 16-entry read buffer was used to eliminate and correct all XOM related problems with computation latency of 20 cycles. RUNMAC was implemented on Java Development Kit under the platform of ASP.Net framework. To evaluate the performance impact of RUNMAC as against XOM, five integer benchmark simulations were carried out by adjusting memory values on RUNMAC and XOM. The result showed that encryption unit in XOM degrades performance by increasing the memory access latency. The performance result showed that RUNMAC used average of 6.4s to detect unauthorized foreign codes as against 11.0s that is common with XOM on the benchmark programs, which was reduced to less than 5s by increasing the hash size of the instruction cache. With the evaluated results, RUNMAC demonstrated an improved precision through several program test codes. This showed that RUNMAC can detect flow anomalies that are common with XOM architecture and protect against unauthorized codes using a new layer of defence concurrently with existing security tools.

Keywords: Intrusion Detection System, Active Network, Hash function, Encryption

Introduction

Intrusion detection systems have existed for a very long time. In the natural world, we have night watchmen and guard dogs. In this case, they serve two purposes: they provide a means of identifying that something bad was happen-

ing and they provided a deterrent to the penetrator. Burglar alarms could also be seen as a form of an intrusion detection system. If the alarm system detects an event that it is programmed to notice, the alarm sounds and it could even give a notice to the police.

All these examples share a single, principal aim: detect any attempt to penetrate the security perimeter of the item (car, house etc.) being secured. If we translate the concept of the alarm system to the computer-networking world, we have an Intrusion Detection System (IDS). An Intrusion Detection System is the generic term given to any hardware, software, or combination of the two that monitors a system or network of systems looking for suspicious activity.

A host-based intrusion detection system (HIDS) monitors a process' execution to identify potentially malicious behaviour. In a model-based anomaly HIDS or behaviour-based HIDS (Debar et al., 1999), deviations from a pre-computed model of expected behaviour indicate possible intrusion attempts. An execution monitor verifies a stream of events, often system calls, generated by the executing process. The monitor rejects event streams deviating from the model. The ability of the system to detect attacks with few or zero false alarms relies entirely upon the precision of the model.

The increasing complexity of modern computer systems has also contributed to the increase in computer security vulnerabilities. The most dangerous type of vulnerabilities allows an attacker to cause program flow anomalies during program execution, leading to arbitrary code execution on the victim computer (SANS, 2006). Many of the most disruptive network worms recently encountered (e.g. Blaster, Slammer, Code Red, Nimda) have exploited such vulnerabilities (Kienzle, 2003; SANS, 2006). Malicious code of this kind can propagate very fast and cause severe network disruption and data loss even before it can be identified (Kienzle, 2003). For example, the Slammer network worm (released January 2003) infected more than 90% of all the vulnerable systems in under 10 minutes, before any meaningful human response was possible (Kienzle, 2003; Moore et al., 2003).

A number of research ideas in the area of network programming can be grouped under the general heading of "active networks" (Collier, 1998). The term active networks arises from the work of Tennenhouse's group in MIT (Tennenhouse, 1996), and the associated ARPA-funded research programme. Security and authentication are major concerns because the level of code penetration in the network is potentially wider (Collier, 1998).

Security threats related to unauthorized code include: viruses (excluding macro viruses); Trojan horses; spyware and adware (programs that monitor system activity such as browsing habits and display unsolicited ads); and backdoor programs used in Distributed Denial of Service (DDoS) attacks. Clearly, a reliable mechanism to detect and prevent unauthorized code execution will contribute significantly to computer security.

In this paper, we describe an architectural technique, which we call Runtime Monitoring of Malicious Codes in a Network (RUNMAC) built around Fiskiran (2004) to monitor program execution and to detect flow anomalies that may be linked to malicious code execution. The key idea in RUNMAC is the real-time verification of program code at the hash block (similar to a basic block) level. Therefore, RUNMAC can detect program flow anomalies that occur during execution such as buffer overrun attacks commonly used by network and email worms.

Related Works

Several intrusion detection systems have been proposed and implemented. Most of them derive from the statistical intrusion detection model of Dorothy Denning (1987). Some of them, for example NIDX (Bauer, 1988), Haystack (Smaha, 1988), IDES (Lunt et al., 1989), MIDAS (Sebring et al., 1988), Wisdom and Sense (Liepins, 1989) and CMDS (Proctor, 1994) use the audit trail

generated by a C2 or higher rated computer, for input. Others, for example NICE (Maccabe & Heady, 1990) and NSM (Heberlein et al, 1991) try to analyze intrusions by analyzing network connections and the flow of information in a network. Others still, such as DIDS (Snapp, 1991) have expanded the scope of detection by distributing anomaly detection across a heterogeneous network and centrally analyzing partial results of these distributed sources to detect potential intrusions that may be missed by the individual analysis of each source.

Among non-statistical approaches to intrusion detection is the work by (Teng, 1990) that analyzes individual user audit trails and attempts to infer the sequential relationships between events; and the neural net modelling of behaviour by (Fox et al., 1990).

Ko et al. (1994) distinguished their work by trying to limit the damage caused by errors in privileged programs while others are trying to limit the damage caused by a Trojan horse or virus. The work monitors the execution of privilege programs to detect flaws in the implementation of the programs from their source code using slicing, dataflow coverage metrics, and symbolic evaluation techniques. The system failed to address other system components in active networks like DNS, NFS, and routers which are taking care in the system proposed.

According to Fiskiran (2004) subsequent studies expanding on detecting malicious code are presented in (Gassend, 2003; Lie, 2000, 2003; Maude, 1984; Suh, 2003; Yang, 2003). We focus on the eXecute Only Memory (XOM) and the memory integrity verification architectures described in Lie (2000) and Gassend (2003). In the XOM architecture, software is distributed in encrypted form by the vendor and decrypted during execution on the target processor using a secret key. The encryption/decryption unit is between the cache and the main memory. Because the memory is untrusted, integrity verification is also required. This is done by tagging each memory block with a keyed hash (HMAC) (Lie, 2000; Menezes, 1996). One of the shortcomings of XOM is that it does not completely protect against an attacker tampering with memory (in particular against replay attacks) (Menezes, 1996). To address this problem, the memory hashing (MH) scheme in Gassend (2003) was proposed. While XOM and MH architectures provide important security functions, they have several shortcomings that limit their usefulness. First, XOM only protects encrypted code whereas most of today's software is unencrypted and a significant fraction is open source. Second, XOM does not protect shared library code, which always exists in plaintext form, whereas virtually all modern applications rely on shared code. Third, neither XOM nor MH fully protects untrusted I/O channels, such as network interfaces. Fourth, none reliably detects flow anomalies that happen during program execution, which is typical of malicious code activity.

There are also host-based and network-based Intrusion Detection (ID) tools to detect anomalous system and network activity. Examples include Stames (2000) and Zou (2003). In general, ID tools can only identify intrusions with a delay (Kemmerer, 2002), which is often significant. Therefore, their usefulness is limited against fast-propagating malicious code. A close study is done by Lie et al. (2000) which we intend to address by adding new layer of Elliptic Curve (EC) based Multi-signcryption and code mobility in the areas of security, authentication, and resource discovery as regard active networks.

RUNMAC Model Design

RUNMAC verifies program execution at the basic block (hash block) level. We define a hash block as "a sequence of instructions with a single entry point, single exit point, and no internal flow control instructions, such as branch, call, and return instructions". While this is generally identical to the definition of a basic block, we prefer to define a hash block explicitly since the basic block definition occasionally excludes the "single entry point" requirement.

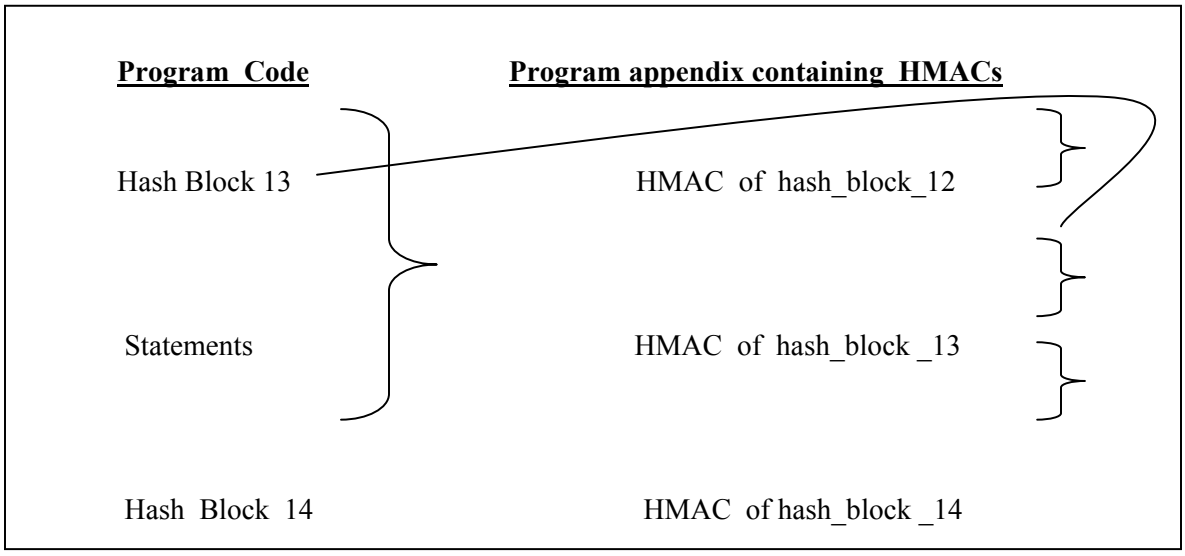
RUNMAC involves computing an HMAC (keyed hash) for each hash block of a program when it is first installed on the host computer. The HMACs are then appended to the program as illus-

trated in figure 1. A new instruction, hash_ptr (hash pointer), is added to the expansion slot. Each hash block begins with a hash_ptr instruction, whose immediate operand points to the corresponding HMAC in the program appendix. This facilitates finding the HMAC of a given hash block during execution. Since the generation of the HMACs and the insertion of hash_ptr instructions can be performed directly on executable code, recompilation and compiler modifications are not necessary. This makes RUNMAC suitable for protecting proprietary and legacy code, where source code is not available.

The key used to generate the HMACs is called the RUNMAC key, which is randomly assigned to each processor and is required for software installation. HMACs can be generated using a hash algorithm or a symmetric-key cipher. In this paper, we use the Elliptic Curve (EC) based Multi-signcryption scheme with 128-bit keys because it has a minimal memory utilization and very fast in hardware implementations (Ajayi, 2009). We set the default HMAC size equal to the AES block size, which is 128 bits. To compute the HMAC of a hash block, we first parse the instructions into 128-bit blocks (i.e. groups of four if the instruction size is 32-bits). Zero padding is used if a block contains uneven number of instructions.

Each block is then encrypted with AES using the RUNMAC key. Finally, the encrypted blocks are XOR'ed together to generate a 128-bit HMAC. Using AES in the ECB mode is acceptable because the hash block size is usually small (Menezes, 1996). Therefore, we believe that the security of this scheme is not less than other 128-bit hash algorithms.

There are many hardware designs for fast AES implementation. One example is described in Kuo (2001), which can perform AES encryption in 10 cycles, with an effective pipelined latency of 1 cycle. The area of this design is reported as 173,000 gates. Better performance (or smaller area) can be realized for RUNMAC by exploiting the fact that the RUNMAC key is fixed for each processor. In this dissertation, we will assume using an AES unit with a 20-cycle absolute latency and a 1-cycle effective pipelined latency.



Operand of the leading hash-pointer instruction points to starting address of the corresponding HMAC

Figure 1: Program Code on RUNMAC

RUNMAC Architecture

The data path for RUNMAC architecture is illustrated in Figure 2. The HMAC Compute Logic (HCL) reads instructions in 128-bit blocks and computes the HMAC corresponding to the current

hash block. HCL is connected to the pipeline control and processes an instruction block only after all the instructions in the block are committed. This simplifies the handling of instructions that are speculatively issued and instructions issued in branch delay slot(s), when these may be conditionally nullified. HCL also interfaces to the cache to save (restore) its internal state on context switches and interrupts.

Concurrent to the HMAC computation, the stored HMAC of the current hash block is read from the memory and stored in the first-in-first-out (FIFO) hash read buffer. This buffer is necessary because the HMAC computation latency is longer than the HMAC lookup latency. The address of the HMAC corresponding to the current hash block can be computed simply by scaling the operand of the leading hash_ptr instruction, and then adding this value to the starting address of the HMAC appendix.

When the HMAC computation is finished, it is compared to the stored HMAC, and an exception is raised if the values mismatch.

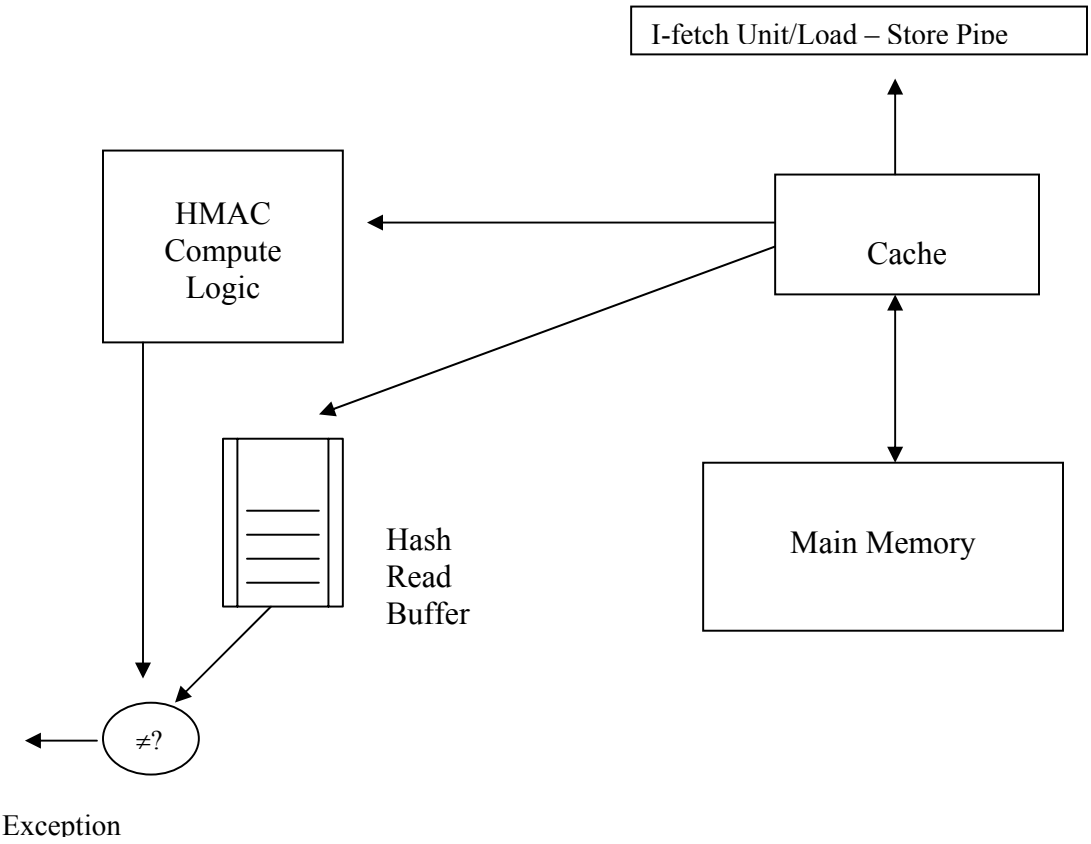


Figure 2: RUNMAC Data Path Architecture

RUNMAC Algorithm

HMAC is a hash based MAC algorithm defined in RFC 2104. It can use any hash function such as SHA1 which we called H. HMAC also requires a user supplied secret key, which is a string of bytes of any length.

The hash algorithm H has two important properties which were fed into the algorithm. The first is the hash size, L. For example MD5 has a hash size of 128 bits (16 bytes). The second quantity is slightly less obvious - it is the block size B of the iterated hash. In general B is greater than L.

Normalizing the Key Length

The first stage of the algorithm is to convert the key to be exactly B bytes long. If the key length is less than B bytes, this is done by adding zero bytes to the end of the key, to form K of exactly B bytes.

However, if the key has more than B bytes to start with, first hash it using H. Then pad the hash value with zeros to make K (again, exactly B bytes).

Creating the Inner and Outer Keys

Now create 2 variants of K, by a simple XOR procedure:

The inner key, K_i is formed from K by XORing each byte with memory address 0x36.

The outer key, K_o is formed from K by XORing each byte with memory address 0x5C.

Calculating the MAC

We use the notation H(x) to represent the hash of byte sequence x. We use H(x, y) to represent the hash of the concatenation of byte sequence x followed by y. Then the MAC of message m is:

$$H(K_o, H(K_i, m))$$

In other words concatenate the inner key with the message, and calculate the hash. Then concatenate the outer key with the hash value and calculate the hash of that.

This method creates a MAC of length L (the hash size of H). It is possible to create a shorter MAC, if required, by truncating the MAC to t bits. To do this simply use the leftmost t bits and discards the remainder. The HMAC specification recommends that t should not be less than half of L, and in any case should never be less than 80, otherwise the MAC might not be secure.

Choosing a Key

The initial key can be any byte sequence of any length. Ideally it should be a random sequence, generated by a cryptographically strong random number generator. For the sake of security, it should not be less than L bytes. However, there is probably not a great deal to be gained by making the key larger than L, and certainly there is no point making it larger than B because then it will simply be hashed back down to L bytes.

If you are using password or passphrase, the situation is different because an L character password is much less random. There is an advantage in using larger passwords, even phrases which are larger than B (even though this will be hashed down to L bytes, a longer the pass phrase will create more randomness in the final L bytes). Of course in practical terms a password of 32 characters or more can start to become cumbersome.

Key Derivation

Most algorithms which permit user selected keys (such as symmetric encryption and MACs) require a binary key, typically 128 bits or more. This equates to a hexadecimal string of at least 32 characters. Most of us would struggle to remember such a key, or indeed to type it in accurately.

Generally most of us prefer using a password rather than a long binary key. The process of converting the password into a binary key is known as Key Derivation.

There are several ways to derive a key from a password, the most common being hash functions and psuedo-random number generators.

Iterative Hash Functions

Most common hash algorithms are block based, and rely on a compression function C. The compression function has a block size B, and an output size L (which also corresponds to the hash size).

The compression function looks like this:

$$z = C(x, y)$$

Here x is a quantity of length B bits, y is a quantity of length L bits, and the result z contains L bits. C is generally quite a complex function for which any small change in x or y creates a large change in z. It is called a compression function simply because it reduces a larger quantity of bits (B+L) to a smaller quantity, L bits (it has nothing to do with ZIP compression, you certainly cannot reverse the function to find a and b from x).

To calculate the hash of a message, the message is divided into n blocks m_0 to m_{n-1}, each of size B bits. If necessary the data is padded to form complete blocks (more below). The compression function is applied iteratively:

$h_0 = C(m_0, IV)$

$h_i = C(m_i, h_{i-1})$

$h_{n-1} = C(m_{n-1}, h_{n-2})$

The initial value IV is a fixed value (it is algorithm specific) - it is L bits long. The hash value is the final output h_{n-1}.

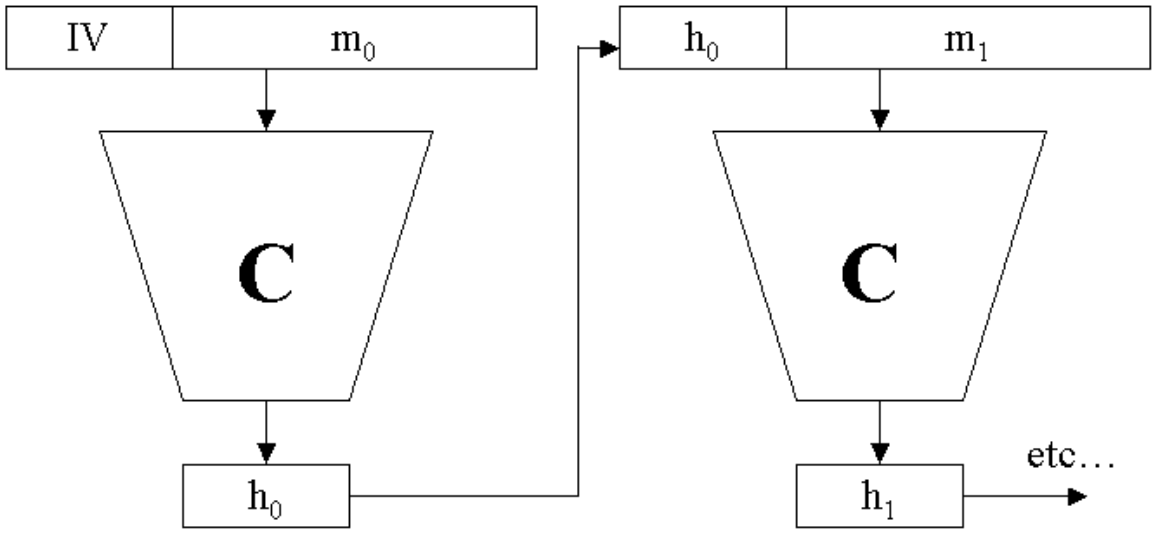


Figure 3: RUNMAC Algorithm Structure

In order to calculate the hash for a stream of bits of arbitrary size, the data must be padded to an exact multiple of the block size. Padding is algorithm specific. The most common scheme if fairly simple – append a single 1 bit, a number of 0 bits, followed by the size of the unpadded message in bits (as a 64 bit integer), such that the padded message is an exact multiple of B bits. This

scheme protects against the cases where very similar messages of slightly different length might have the same hash.

Security Approach

RUNMAC can detect flow anomalies that occur during program execution. Most of the research work into security is concentrating on the malicious issue, by advancing techniques that isolate the execution of malicious code from the rest of the system. However isolating on its own is only a first step for security. A security framework must furnish further properties. It is important that an agent that visits a trustworthy host must be able to authenticate the information that it furnishes.

The core of the security approach is called EC Multi-signcryption scheme that provides integrity for the stored HMAC. The EC Multi-signcryption scheme is a cryptographic method that fulfils both the functions of secure encryption and digital multi-signature for multi-users in a network. The RUNMAC architecture using the EC Multisigncryption provides memory integrity verification and program code confidentiality. In our opinion, the exciting future work on this research is to take advantage of the mobility nature of active networks to make RUNMAC to provide memory integrity and program code confidentiality using secured agents.

Discussion

We presented a novel approach to intrusion detection by verifying program code at the hash block level to stop progressing intrusion at early stages. In addition, the RUNMAC monitor instructions whose behaviour is typically exploited by malicious code. Below we discuss our work and compare it with other approaches.

Comparison with Code Encryption and Integrity Checking – XOM Architecture

The line of research that most closely parallels this work is the code encryption and integrity verification methods proposed for Digital Rights Management. The work focused on the eXecute Only Memory (XOM) described in (Lie, 2003). In the XOM architecture, software is distributed in encrypted form by the vendor and decrypted during execution on the target processor using a secret key.

One of the shortcomings of XOM is that it does not completely protect against an attacker tampering with memory (in particular against replay attacks) which the work addressed. While XOM architectures provide important security functions, they have several shortcomings that limit their usefulness. First, XOM only protects encrypted code whereas most of today's software is unencrypted and a significant fraction is open source. Second, XOM does not protect shared library code, which always exists in plaintext form, whereas virtually all modern applications rely on shared code. Third, XOM does not fully protect untrusted I/O channels, such as network interfaces. Fourth, the architecture does not reliably detect flow anomalies that happen during program execution, which is typical of malicious code activity that RUNMAC addresses. Finally, the execution time of RUNMAC is less than that of XOM in performing intrusion activities.

Comparison to Misuse Detection

In misuse detection, the goal is to identify actions (or misuse signatures) that represent intrusive activities and to check for occurrences of these actions in the audit trails. Misuse signatures are described by expert-system rules, state-transition diagrams, and patterns in Petri networks.

The specification-based approach can be thought of as the dual of misuse detection.

A misuse signature describes undesired behaviour in a system while a trace policy describes the desirable behaviour of a subject. In particular, our approach focuses on the desirable behaviour of unauthorized foreign code on the victim computer. On way to specify the desirable behaviour of a program is to enumerate the operations the program needs to perform in order to accomplish its function.

A misuse detector matches a signature with the whole system trace to identify intrusions while an analyzer in a specification-based execution monitor parses the trace of a subject to determine whether the subject conforms to a trace policy. Although matching of different signatures can be distributed over multiple hosts, each misuse detector requires the whole system trace. In a distributed system with many hosts, the whole system trace would be huge and cannot be processed by a misuse detector in real time. In our approach, the RUNMAC detect program flow anomalies by verifying the code at the hash block level which can be done in real time.

In misuse detection, signatures are mostly driven by previous attacks or known vulnerabilities. Although possible, it is not intuitive to encode a policy as misuse signatures. Our approach is more policy-oriented; a trace policy for a subject is specified based on the functionality of the subject and the system security policy. Therefore, it can succeed in catching attacks that exploit unknown vulnerabilities in programs.

Comparison to Type Enforcement

In the type-enforcement approach (Badger, 1995), accesses to objects by a subject are restricted by a type-enforcement policy based on the domain of the subject and the type of the object. Each subject is running in a domain and each object is assigned to a fixed type when it is created.

The Domain and Type Enforcement (DTE) approach (Badger, 1995) applies type enforcement to a Unix system. It takes the process hierarchy and the file hierarchy of current systems into consideration. The type enforcement policy is specified in a DTE language. Each domain is associated with one or more entrance programs, when executed by a subject/process switches, will move the subject/process to that domain. In effect, the type enforcement policy restricts the access of a process based on the program it is executing.

The DTE approach is similar to our approach as it further restricts the accesses of a program. In general, a DTE policy can be specified by a set of trace policies in our approach. A trace policy can specify the valid accesses of a program, but also the valid ordering of the accesses. Therefore, a trace policy is more expressive than a DTE policy regarding the specification of the behaviour of a program.

One difference between DTE and our approach is that DTE is a preventive approach.

Operations performed by a program during execution that violate the DTE policy are denied by the DTE subsystem, while ours is a detection approach that raises a warning when a violation occurs. Nevertheless, one can incorporate our parsing mechanism into a reference monitor that prohibits any operations that are not accepted by the parser (i.e., those operations that are in violation of a trace policy).

Performance

To evaluate the performance impact of the RUNMAC architecture, simulations were carried out and the results verified as described in the subsequent section.

Code Size

The RUNMAC architecture increases the program size due to the hash_ptr instructions and the appended HMACs. Figure 4 shows the total storage overhead when 32-bit, 64-bit, 128-bit HMACs are used. The total overhead is increasing in the order for 32-bit, 64-bit, and 128-bit HMACs respectively. While that of the 128-bit HMACs is high, the total size increase on a system may be limited by using RUNMAC only on vulnerable applications, such as webservers and mailservers that maintain continuous network connections. For resource-constrained environments, smaller 32-bit HMACs may be preferred to limit the total overhead size.

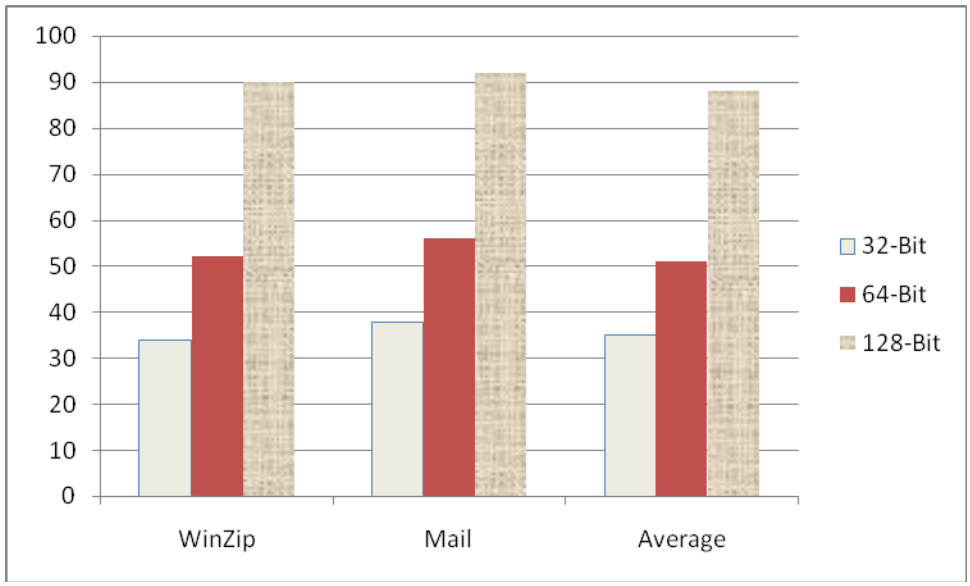


Figure 4: Code Size

Efficiency Analysis

We evaluated RUNMAC as against XOM using five integer benchmark simulations by adjusting the memory values. The result showed that performance of RUNMAC averages 6.4s as against 11.0s in detecting unauthorized foreign codes (as shown in figure 5). By increasing the hash size of the instruction cache, the average time was reduced to less than 5s which suggested that RUNMAC demonstrated an improved precision and can detect flow anomalies in a very short time.

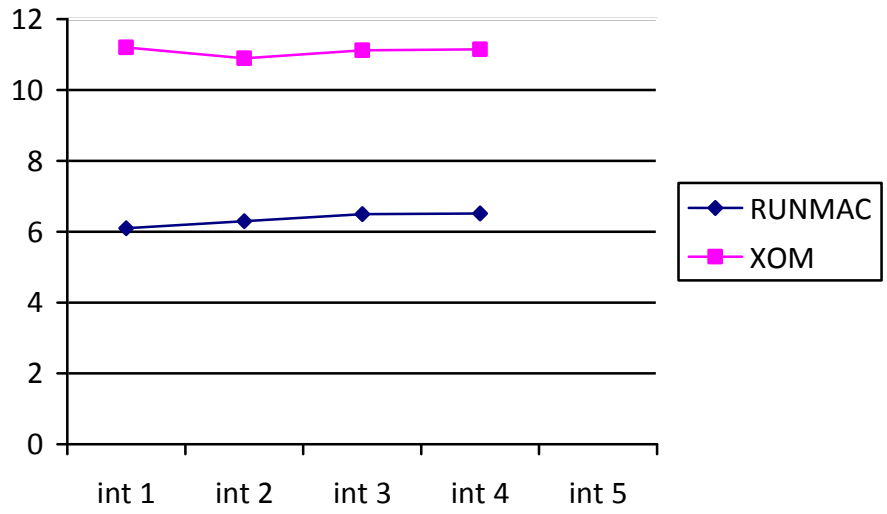


Figure 5: Integer Benchmark Simulation Results

Conclusions

The approach has been to develop an architectural technique, which is called an Execution Monitoring of Remote Intrusions, to detect program/data flow anomalies associated with unauthorized code execution on computer workstations. The approach is based on a model that verifies program code or data at the hash block level. By verifying at such level, it can monitor instructions whose behaviour is typically exploited by malicious code, such as branch, call, return instructions. While it is not a one-size-fits-all solution against all such malicious code, it can contribute significantly to network security if it is employed as a new layer of defence concurrently with existing security tools.

References

Ajayi, O. B., Akinde, A. D., & Akinwale, A. T. (2009). Framework on hybrid network management system using a secure mobile agent protocol. *Issues in Informing Science and Information Technology, 6*.

American Bankers Association [ABA]. (2000). *Keyed hash message authentication code, ANSI X9.71*. Washington, D.C.

Anderson, J. P. (1980). *Computer security threat monitoring and surveillance*. [Technical report]. James P. Anderson Company, Fort Washington, Pennsylvania.

Ashcraft, K., & Engler, D. (2002). *Using programmer-written compiler extensions to catch security holes*. In IEEE Symposium on Security and Privacy, Oakland, CA.

Badger, L. et al. (1995). Practical domain and type enforcement for UNIX. *Proceedings of the 1995 Symposium on Security and Privacy*, Oakland, CA, pp. 66-77.

Ball, T., & Rajamani, S. K. (2001). The SLAM toolkit. *13th Conference on Computer Aided Verification,* pages 260-264, Paris, France.

Baratloo, A., Tsai, T., & Singh, N. (2000). Transparent run-time defence against stack smashing attacks. *USENIX Annual Technical Conference*, San Diego, CA.

Bauer, D. S., & Koblentz, M. E. (1998). NIDX - An expert system for real-time network intrusion detection. *Proceedings of the Computer Networking Symposium*, pages 98-106. IEEE, New York, New York.

Beattie, S. M., Black, A. P., Cowan, C., Pu, C., & Yang, L. P. (2000). *Cryptomark: Locking the stable door ahead of the Trojan Horse*. [Technical review.]

Best, R. M. (1980). Preventing software piracy with crypto-microprocessors. *Proceedings of IEEE COMPCON*, pp. 466-469.

Browne, H. K., Arbaugh, W. A., Hugh J. M., & Fithen, W. L. (2001). A trend analysis of exploitations. *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pp. 214-229.

Bush, W. R., Pincus, J. D., & Siela, D. J. (2000). A static analyzer for finding dynamic programming errors. *Software Practice and Experience, 30*(7),775-802.

CERT® Advisory CA-2000-04. (2000). *Love letter worm.*

CERT® Advisory CA-2001-19. (2001). *"Code Red" worm exploiting buffer overflow in IIS indexing service DLL.*

Cheeseman, P., Kelly, K., Self, M., Stutz, J., Taylor, W., & Freeman, D. (1988). Autoclass: A Bayesian classification system. *Proceedings of the Fifth International Conference on Machine Learning*, pp. 54-64. Morgan Kaufmann.

Cheeseman, P., Hanson, R., & Stutz, J. (1991). Bayesian classification with correlation and inheritance. *12th International Joint Conference on Artificial Intelligence.*

Chen, H., Dean, D., & Wagner, D. (2004). Model checking one million lines of C code. *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA.

Chiueh, T., & Hsu, F. (2001). RAD: A compile-time solution to buffer over-flow attacks. *21st International Conference on Distributed Computing Systems, Phoenix, AZ.*

Collier, M. (1998). *Mobile agents and active networks: Complementary or competing technologies*. A technical report, Dublin City University, Ireland.

Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., & Zhang, Q. (1998). StackGuard: Automatic adaptive detection and prevention of buffer-over flow attacks. *7th USENIX Security Symposium, San Antonio, TX.*

Cowan, C., Wagle, P., Pu, C., Beattie, C., and Walpole, J. (2000). Buffer overflows: Attacks and defenses for the vulnerability of the decade. *DARPA Information Survivability Conference and Expo (DISCEX*), volume 2, pages 119-129, Hilton Head, SC.

Crosbie, M., & Spafford, E. (1995). Active defense of a computer system using autonomous agents. *Proceedings of the 18th National Information Systems Security Conference*, pp. 549-558.

Debar, H., Dacier, M., & Wespi, A. (1999). Towards taxonomy of intrusion-detection systems. *Computer Networks, 31,* 805-822.

Denning, D. E. (1987). An intrusion-detection model. *IEEE Transactions on Software Engineering, 13*(2).

Detlefs, D. L., Rustan K., Leino, M., Nelson, G., & Saxe, J. B. (1998). *Extended static checking.* Technical Report SRC-159, Compaq SRC.

Dhurjati, D., Kowshik, S., Adve, V., & Lattner, C. (2003). Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES),* pp. 69-80, San Diego, CA.

Diego M. Z. (1996). SAINT: A security analysis integration tool. *Proceedings of the Systems Administration, Networking and Security Conference*.

Doorn, L. V., Ballintijn, G., & Arbaugh, W. (2007). *Signed executables for Linux.*

Dor, N., Rodeh, M., & Sagiv, M. (2001). Cleanness checking of string manipulations in C programs via integer analysis. *Static Analysis Symposium*, volume 2126 of Lecture Notes in Computer Science, page 194, Paris, France, Springer.

DynamoRIO. http://cag.lcs.mit.edu/dynamorio/.

Evans, D. (1996). Static detection of dynamic memory errors. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 44-53, Philadelphia, PA.

Fiskiran, A. M., & Lee, R. B. (2004). *Runtime execution monitoring (REM) to detect and prevent malicious code execution*. Department of Electrical Engineering, Princeton University.

Fox, K. L., Henning, R. R., Reed, J. H., & Simonian, R. (1990). A neural network approach towards intrusion detection. *Proceedings of the 13th National Computer Security Conference*, pp. 125-134, Washington, DC.

Gassend, B., Suh, G. E., Clarke, D., Dijk, M. V., & Devadas, S. (2003). Caches and hash trees for efficient memory integrity verification. *Proceedings of the Internatoinal Symposium on High-Performance Computer Architecture (HPCA),* pp. 295-306.

Habra, N., Charlier, B. L., Mounji, A., & Mathieu, A. (1992). ASAX: Software architecture and rule-based language for universal audit trail analysis. *Proceedings of ESORICS 92*, Toulouse, France.

Haugh E., & Bishop, M. (2003). Testing C programs for buffer overflow vulnerabilities. *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA.

Heady, R., Luger, G., Maccabe, A., & Servilla, M. (1990). *The architecture of a network level intrusion detection system.* Technical Report, Department of Computer Science, University of New Mexico.

Heberlein, L. T., Levitt, K. N., & Mukherjee, B. (1991). A method to detect intrusive activity in a networked environment. *Proceedings of the 14th National Computer Security Conference*, pp. 362-371.

Hofmeyr, S. A., Forrest, S., & Somayaji, A. (1998). *Intrusion detection using sequences of system calls*. Dept. of Computer Science, University of New Mexico.

Ilgun, K., Kermmerer, R., & Porras, P. (1995). State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering, 21*(3).

Ilgun, K. (1993). *USTAT - A real-time intrusion detection system for UNIX*. Technical report TRCS93-26, Computer Science Department, University of California at Santa Barbara.

Intel, IA-32 Intel architecture software developer's manual Volume 2A (2004). Instruction set reference, A-M, available at <http://www.intel.com>.

Kaufman C., Perlman, R., & Speciner M. (1995). *Network security*. Englewood Cliffs, New Jersey: Prentice Hall.

Kemmerer, R. A., & Vigna, G. (2002). Intrusion detection: A brief history and overview. *Computer, 35*(4), 27-30.

Kienzle, D. M., & Elder, M. C. (2003). Recent worms: A survey and trends. *Proceedings of the ACM Workshop on Rapid Malcode*, pp. 1-10.

Kim, G. H., & Spafford, E. H. (1993). *The design and Implementation of TRIPWIRE: A file system integrity checker.* Technical Report, TR-93-071.

Kim, G. H., & Spafford, E. H. (1994). Experiences with Tripwire: Using integrity checkers for intrusion detection. *System Administration, Networking and Security Conference III: USENIX '94*.

Kiriansky, V., Bruening, D., & Amarasinghe, S. P. (2002). Secure execution via program shepherding. *11th USENIX Security Symposium,* pp. 191-206, San Fancisco, CA.

Ko, C., Fink, G., & Levitt, K. (1994). Automated detection of vulnerabilities in privileged programs by execution monitoring. *Proceedings of IEEE.*

Kowshik, S., Dhurjati, D., & Adve, V. (2002). Ensuring code safety without runtime checks for real-time control systems. *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 288-297, Grenoble, France.

Krawczyk, H., Bellare, M., & Canetti, R. (1997). *HMAC: Keyed-hashing for message authentication*. Internet Engineering Task Force, Request for Comments (RFC) 2104.

Kuo, H., & Verbauwhede, I. (2001). Architectural optimization for a 1.82 Gb/s VLSI implementation of the AES Rijndael algoritm. *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Lecture Notes in Computer Science, vol. 2162, pp. 51-64.

Lankewicz, L. (1992). *A non-parametric pattern recognition to anomaly detection.* Ph.D. Thesis, Tulane University, Department of Computer Science.

Larochelle, D., & Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities. *10th USENIX Security Symposium, Washington, DC.*

Larson, E., & Austin, T. (2003). High coverage detection of input-related security faults. *12th USENIX Security Symposium, Washington, DC*.

Lehti, R., & Virolainen, P. (2007). *AIDE (Advanced Intrusion Detection Environment)*. Retrieved from http://www.cs.tut.fi/~rammer/aide.html

Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., & Horowitz, M. (2000). Architectural support for copy and tamper resistant software. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 168-177.

Lie, D., Mitchell, J., Thekkath, C. A., & Horowitz, M. (2003). Specifying and verifying hardware for tamper-resistant software. *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 166-177.

Liepins, G. E., & Vaccaro, H. S. (1989). Anomaly detection: Purpose and framework. *Proceedings of the 12th National Computer Security Conference*, pp. 495-504.

Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R., & Zissman, M. (2000). Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. *Proceedings of DARPA information survivability Conference and Exposition*.

Lippmann, R., Fried, D., Haines, J., Corba, J. and Das K. (2000). Evaluating intrusion detection systems: Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. *Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection (RAID 2000)*.

Livshits, V. B., & Lam, M. S. (2003). Tracking pointers with path and context sensitivity for bug detection in C programs. *European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pp. 317-326, Helsinki, Finland.

Lunt, T. F., Jagannathan, R., Lee, R., Whitehurst, A., & Listgarten, S. (1989). Knowledge based intrusion detection. *Proceedings of the Annual AI Systems in Government Conference, Washington, DC.*

Maccabe, A. B., McDonald, R., & Anand, V. (2008). *Google search - Learning how to characterize normal behavior in local area networks*.

Maude, T., & Maude, D. (1984). Hardware protection against software piracy. *Communications of the ACM, 27*(9), 950-959.

Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of applied cryptography*. CRC Press.

Moitra, A. (2007). *Real-time audit log viewer and analyzer*.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., & Weaver, N. (2003). Inside the Slammer Worm. *IEEE Security and Privacy Magazine, 1*(4), 33-39.

National Institute of Standards and Technology. (1995). *Secure hash standard (SHS)*. Federal Information processing Standards Publication 180-1, 17.

National Institute of Standards and Technology. (2001). *Advanced encryption standard (AES)*. FIPS Pub. 197. Retrieved from http://csrc.nist.gov/publications/fips

National Institute of Standards and Technology. (2001). *Security requirements for cryptographic modules*. Federal Information Processing Standards Publication 140-2.

Ozdoganoglu, H., Brodley, C. E., Vijaykumar, T. N., Kuperman, B. A., & Jalote, A. (2002). *SmashGuard: A hardware solution to prevent security attacks on the function return address*. Technical Report TR-ECE 03-13, School of Electrical and Computer Engineering, Purdue University.

Porras, P. A., & Kemmerer, K. A. (1992). Penetration state transition analysis: A rule-based intrusion detection approach. *Eighth Annual Computer Security Applications Conference*, pp. 220-229. IEEE Computer Society press, IEEE Computer Society press.

Proctor, P. (1994). *Audit reduction and computer misuse detection*. Talk given at the Sixth Annual Computer Security Incident Handling Workshop.

Sandeep K., & Eugene H. S. (1994). A pattern matching model for misuse intrusion detection. *Proceedings of the 17th National Computer Security Conference*, pp. 11-21.

Sebring, M., Shellhouse, E., Hanna, M., & Whitehurst, R. (1988). Expert systems in intrusion detection: A case study. *Proceedings of the 11th National Computer Security Conference*.

Security Focus, Mailing List: FOCUS-IDS, (2006) Retrieved from http://www.securityfocus.com/archive/

Smaha, S. E. (1988). Haystack: An intrusion detection system. *Fourth Aerospace Computer Security Applications Conference*, pp. 37-44, Tracor Applied Science Inc., Austin, Texas.

Smaha, S. E., & Winslow, J. (1994). Misuse detection tools. *Computer Security Journal, 10*(1, Spring), 39-49.

Smaha, S. E. (1995). Talk given at the third *Computer Misuse and Anomaly Detection Workshop (CMAD III*) in Sonoma, California.

Snapp, S. R., Brentano, J., Dias, G. B., Goan, D. L., Heberlein, L. T., Ho, C., Levitt, K. N., Mukherjee, B., Smaha, S. E., Grance, T., Teal, D. M., & Mansur, D. (1991). DIDS (Distributed Intrusion Detection System) - Motivation, architecture, and an early prototype. *Proceedings of the 14th National Computer Security Conference*, pp. 167-176.

Spafford, E. H. (1991). *The internet worm incident*. Technical Report CSD-TR-933.

Stames, W. W. (2000). Integrity assessment tools: Fundamental protection for business critical systems, data, and applications. *Proceedings of the International Conference Information Technology Interfaces (ITI)*, pp. 465-470.

Staniford, S., Paxson, V., & Weaver, N. (2002). How to own the internet in your spare time. *Proceedings of the 11th USENIX Security Symposium*.

Suh, G. E., Clarke, D., Gassend, B., Dijk, M. V., & Devadas, S. (2003). Efficient memory integrity verification and encryption for secure processors. *Proceedings of the Annual IEEE/ACM International Symposium on Micro architecture (MICRO)*, pp. 339-350.

Teng, H. S., Chen, K., & Lu, S. C. (1990). Security audit trail analysis using inductively generated predictive rules. *Proceedings of the Sixth Conference on Artificial Intelligence Applications*, pp. 24-29, Piscataway, New Jersey, IEEE.

Tennenhouse, D. L., & Wetherall, D. J. (1996). Towards active network architecture. *Computer Communications Review, 26*(2).

Vaccaro, H. S., & Liepins, G. E. (1989). Detection of anomalous computer session activity. *Proceedings of the Symposium on Research in Security and Privacy*, pp. 280-289.

Wagner, D., Foster, J. S. Brewer, E. A., & Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. *Symposium on Network and Distributed Systems Security (NDSS)*, pp. 3-17, San Diego, CA.

Wetmore, B. R. (1993). *Paradigms for the reduction of audit trails.* Master's Thesis, University of California, Davis.

Yahalom, R., Klein B., & Beth, T. (1993). Trust relationships in secure systems: A distributed authentication perspective. *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pp. 150-164.

Yang, J., Zhang, Y., & Gao, L. (2003). Fast secure processor for inhibiting software piracy and tampering. *Proceedings of the Annual IEEE/ACM International Symposium on Micro architecture (MICRO)*, pp. 351-360.

Zou, C. C., Gao, L., Gong, W., & Towsley, D. (2003). Monitoring and early warning for internet worms. *Proceedings of the ACM Conf. Computer and Communication Security (CCS)*, pp. 190-199.

Biographies

Ajayi, Ibironke A. studied Computer Science in the University of Ibadan, Ibadan Nigeria. She had M.Sc Computer Science and M.ED in Educational Management. She is presently a Principal Lecturer at the Federal College of Education Abeokuta Nigeria.

Ajayi, Olutayo B. received his postgraduate degrees in Computer Science and at the verge of defending his PhD in Computer Science. He is the Head, Network and internet Services Unit of the University of Agriculture, Abeokuta Nigeria and also a part-time lecturer in the Department of Computer Science at the same University. His research interests include mobile agents, Network Security and Web Applications.