

RECSRF: Novel Technique to Evaluate Program Security using Dynamic Disassembly of Machine Instructions

Asanga Wanniarachchi*, Chandana Gamage*

**Department of Computer Science & Engineering, University of Moratuwa, Sri Lanka*
asanga2605@gmail.com, chandag@cse.mrt.ac.lk

Abstract— Modern software systems have nearly an uncontrolled growth in complex requirements that is fueled by the need to interact with multiple other complex systems. While some of the complexity increases are due to activities such as bug fixes which are unavoidable, other activities such as expanding and enhancing functionality of a software system becomes a risky endeavor to undertake with respect to security as responding to the potential vulnerabilities due to complexity increases require scarce resources including technical expertise and time. This paper introduces RECSRF; a novel framework to quantitatively evaluate the security of an execution in line with the security risk impact it makes over the particular microprocessor on which it executes. RECSRF consists of two components; a novel concept called The Run-time Execution Complexity (REC) of a program execution, which evaluates the trade-off between performance vs. security, while adhering to Control Flow Integrity (CFI) of programs, and an information theoretic technique to approximate the Security Risk Factor (SRF), which approximates the risk of a particular execution by analyzing dynamically disassembled machine instructions of a particular microprocessor. The RECSRF value allows software designers to select the most secure resource combination among a given set of resources, and software implementers to decide whether to proceed or not with a software change. The method can also be used to detect control flow hijacks at runtime by using it as an application level intrusion detection mechanism which allows transforming the same to an application level intrusion preventer upon successful implementation.

Keywords— Software Complexity, Information Security Risk Indexing

I. INTRODUCTION

A core design criteria for many microprocessor families available today was the historical need to support single-user environments. Microprocessor manufactures have been extending the same core design principles even when designing with advanced features such as multi-core, multi-user capabilities and performance improvements. As an example, speculative execution, a special form of out of order execution was introduced to improve efficiency and performance [10]. Over time, microprocessors with such improvements have moved towards multi-user environments such as host virtualized systems despite the fact that the original design principles of the microprocessor were not intended to host systems sharing computing resources. This has created a whole new class of microprocessor level security vulnerabilities with Meltdown [15] and Spectre [14] being the most recent examples of high level of damage possible due to this new vulnerability class.

In complex software systems, formed with smaller modular units without centralized control, local interactions between parts is an emergent feature [17]. The resulting complex program execution environment introduce new problems such as increased maintenance costs, difficulties in testing, and in worst case system back-doors and zero-day vulnerabilities. While there are several software complexity measures to individually evaluate complexity on high level code segments, none of them are addressing the problem of overall complexity that all the program modules collectively enforce on the underlying microprocessor. This gap has

motivated the research work presented in this paper to derive a new risk evaluation framework considering the practical interactions of program execution from the microprocessor's perspective.

II. RELATED WORK

An easy way to comply with the conference paper formatting requirements is to use this document as a template and simply type your text into it. Control-Flow Integrity (CFI) ensures that the execution of a program dynamically follows only certain paths, along with a static policy. It can prevent attacks that exploit buffer over-flows and counter other vulnerabilities such as attempts to control program behavior. Also, there is prior work [8], [9] on the importance of CFI specifically addressing mobile platforms. Information flow analysis has been an active research area in the past few decades [21] and the work of Nicoletta and Luca [11] on instruction level security analysis process works by assigning a security level for each machine instruction. In order to address the challenges associated with control flow graph construction based approaches to study assembly, a non-interference based method has been proposed by Ricardo Medel, et al [4], by defining a set of pseudo-instructions to impose a stack discipline on the control flow of programs. Berlinger [2] developed an information theory complexity measure based on the frequency of program tokens where the method require the grouping of instructions in to tokens. However, it is practically difficult to group assembly instructions this way since some assembly languages will contain several hundred instructions where nearly half of which are not used. There are methods to dynamically monitor operating system kernel integrity based on a property called state-based control-flow integrity (SBCFI) [19] and it is claimed that any violation of SBCFI property will indicate a persistent, unauthorized modification of the control-flow graph of the kernel. Although SBCFI discussion include drawbacks of CFI violations, SBCFI also does not provide a method to quantify risk or a method to address the problem of overall complexity.

III. THEORETICAL BACKGROUND

This section summarizes the theoretical background used in deriving the RECSRF framework..

A. Information Theory and Shannon Entropy

Information theory [2] is concerned with the amount of information contained in a message, or a string of symbols. Claude Shannon provided a scientific definition [22] to information theory with his seminal paper A Mathematical Theory of Communication. Shannon entropy is a key measure in information theory which quantifies the amount of uncertainty involved in the value of a random variable. The entropy of a discrete random variable X is defined by the equation given below.

$$H(X) = - \sum_{x \in X} p(x) \cdot \log p(x) \quad (1)$$

B. Software Complexity

Software complexity is the natural byproduct of the functional complexity that a given code base is attempting to enable [3]. Cyclomatic complexity proposed by McCabe [16] is a major contributor in measuring program complexity, which measures the number of decision making paths in a high level code segment. With high number of moving parts or interfaces between programs and other systems, the system and interfaces become difficult to maintain securely.

1) Cyclomatic Complexity: Cyclomatic complexity [12], [16] is a graph theoretic measure to evaluate the amount of decision logic in a source code function. It is calculated by developing a Control Flow Graph (CFG) [16] of the code that measures the number of linearly-independent paths through a program module. Lower the Cyclomatic Complexity in a program means lower the security risk to and easier to understand. CFG is a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. McCabe provided a graph theoretic definition to cyclomatic complexity and it can be expressed mathematically as below.

$$M = \begin{cases} E - N + P & ; \text{for a subroutine.} \\ E - N + 2P & ; \text{otherwise.} \end{cases} \quad (2)$$

Where, E is the number of edges of the graph, N is the number of nodes of the graph and P is the number of connected components. The Software Engineering Institute (SEI) recommendation on code complexity is shown in Table I.

TABLE I
SEI RECOMMENDATION FOR CYCLOMATIC COMPLEXITY

CC value range	Module risk level
1 – 10	Low
11 – 20	Moderate
21 – 50	High
≥ 51	Very high

2) Entropy vs. Security: Entropy is the concept in information theory, introduced by Shannon [22] as a quantitative measurement of the uncertainty associated with random phenomena. It can be stated that phenomenon A represents less uncertainty than phenomenon B, if the result of experimentation associated with A is more likely to occur than that of B. Information theory software complexity metrics measure the degree to which infrequently used instructions occur in the program. Higher the infrequently used instructions in a program, higher the complexity value assigned to the program by the metric.

C. Conditional Execution in Assembly

Among different categorizations of x86 instructions [6], control transfer instructions reflects the degree to which programs comply with Control Flow Integrity (CFI) [5]. Control transfer instructions has two forms; conditional and unconditional. Conditional control transfers depend on the results of operations that affect the flag register, but unconditional control transfers are always executed. Unconditional Control Transfer (UCT) instructions JMP, CALL, RET, INT and IRET transfer control from one code segment location to another [18]. The JMP instruction always transfer control from the current location to a new location. CALL (Call Procedure) activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction. CALL places the current value of Instruction Pointer Register on the stack. The RET instruction in the called procedure uses this address to transfer control back to the calling program, and IRET returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception.

1) Instruction Execution and CPU State Concept: When executing instructions, the microprocessor will set different CPU flags, register values and will facilitate for all the background needs of the execution, since it has to be on certain state(s) in order to execute instructions securely. The state here is a vital and an influential term that has

been extensively adopted during this research, which is more like a snapshot view of a microprocessor when executing an instruction. This means, depending on register values, flag status and other constraints, the microprocessor will be on different states with different capabilities, which introduce different security risks. For an example, the CPU flags and register values will be completely different when the microprocessor executes a low privileges instructions (such as instructions from user space) and a high privileged instructions (such as instructions from kernel space).

2) The Instruction Pointer Register and its Security Impacts: The role of the instruction Pointer Register is to provide the address of the next instruction to be executed [7]. CPUs Fetch-Execute cycle ensure that instructions are fetched sequentially from memory, but control transfer instructions such as branches, conditional expressions, subroutine calls, and returns will change this sequence by placing arbitrary values in the instruction pointer [18], and this makes branch prediction extremely difficult. We argue that a higher number of control transfer instructions in an instruction trace will open up a whole new class of security risks. For an example goto is an unconditional jump statement, that can be used to jump from anywhere to anywhere in a function. When this happens, the program control will be transferred unconditionally to a new location, which greatly affects the control flow integrity of the program.

3) Cyclomatic Complexity and IP Register Volatility: Increased cyclomatic software complexity of programs means that those programs contains higher number of branches, conditional expressions, subroutine calls etc. These are all presented to the microprocessor as control transfer instructions which adversely affects the control flow integrity of the execution, because once a control transfer instruction is present, the program control will be transferred to an arbitrary memory location specified by the specific instruction caused control transfer.

The memory addresses pointed by the instruction pointer register will not always be in sequential order when higher number of control transfer instructions are present. This makes the instruction pointer register highly volatile in terms of next instruction predictability and we define this as Instruction Pointer (IP) Register Volatility. Accordingly, higher the IP register

volatility means higher the affect to the control flow integrity of the program and this makes the control flow hijacks [23] highly probable.

Therefore, we argue that higher the cyclomatic complexity of software programs is an indication of higher instruction pointer register volatility and as a consequence the control flow integrity of the program will be greatly affected thus making the execution trace vulnerable to control flow hijacks.

IV. THE NOVEL RISK EVALUATION METHOD RECSRF

This research is focused on using instances that introduce state changes to the instruction pointer register of microprocessor as a barometer for measuring volatility of the instruction pointer register during a program execution. The hypothesis is that higher volatility will lower overall security of the system by making accurate branch prediction more difficult and thereby increase the probability of system being vulnerable to control flow hijack attacks [23].

We have analyzed unconditional control transfer instructions in detail, since our intention is to derive a lower bound for the risk. The unconditional control transfer instructions will always be executed, hence by analyzing the disassembled instruction code, it can be stated that at those points the program control has changed. Since the instruction pointer register manipulation itself is a risky operation, we argue that any privilege elevation after this state change will be a potential attack attempt. The method proposed in this research will identify this behavior and will quantitatively measure its risk. RECSRF considers every unconditional control transfer as a potential control flow hijack [23], [24] attack and any return as a malicious return.

We observed that the existing information theoretic methods cannot be used effectively to calculate the actual risk of an execution, since none of them are taking the dynamic nature of interactions of different program modules in to consideration. The Runtime Execution Complexity (REC), a novel extension to the traditional complexity, which accounts practical interactions of program modules rectify this difficulty.

A. The Runtime Execution Complexity (REC) Concept

The Runtime Execution Complexity (REC) is the overall cyclomatic complexity a program will

introduce during its runtime, i.e. the complexity of the main program and all the other program modules that the main program invokes and interacts with. Since the execution flow of the program will be changed at program control transfers, we argue that this control flow change will introduce a security risk to the overall execution and that it will increase the Runtime Execution Complexity. For an example, program A may be calling a library program B and it may be calling another program say B1. However the original programmer of program A may be unaware of program internals of B and B1. It is also possible that program A is well written with less cyclomatic complexity, but B1 may be poorly written and consists of a higher cyclomatic complexity value. However when A is presented to the microprocessor for execution, it is the overall complexity that comes in to effect, and due to the effect of B1, the overall program will be vulnerable to control flow hijacks due to its higher cyclomatic complexity value. The proposed evaluation scheme is given below as a step wise guide.

Step 1: Attach the process ID (PID) of the main program to the disassembled code analysis utility. To optimize instruction collection and to eliminate false positives, make sure that a minimum amount of processes are running in the background.

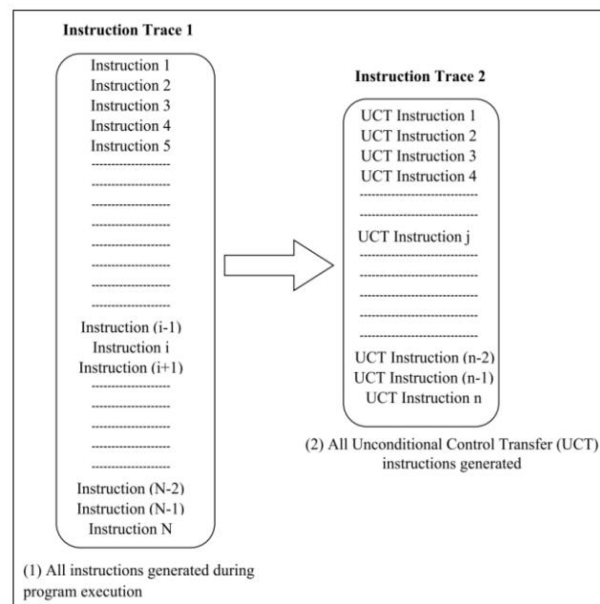


Fig. 1. Mapping of extracted unconditional control transfers to the original instruction trace

Step 2: Extract all the unconditional control transfer instructions and privilege elevation

instructions from the initial instruction trace from Step 1 (see Figure 1).

Step 3: Calculate the number of unconditional control transfer instructions in this extracted instruction trace. Assume there exists n number of unconditional control transfer instructions in the original instruction trace, which means the original instruction trace can be divided into $(n+1)$ number of regions (see Figure 2). We call this region a Threat Block and we argue that these threat blocks will increase the attack surface of the execution. Figure 3 has illustrated this concept. Even though there are $(n+1)$ number of threat blocks, we can safely eliminate the very first block (Block 0 in Figure 2) since it is derived from the main program we trust (we treat it to be inside the Trusted Computing Base (TCB) [20]). This results in having n number of threat blocks.

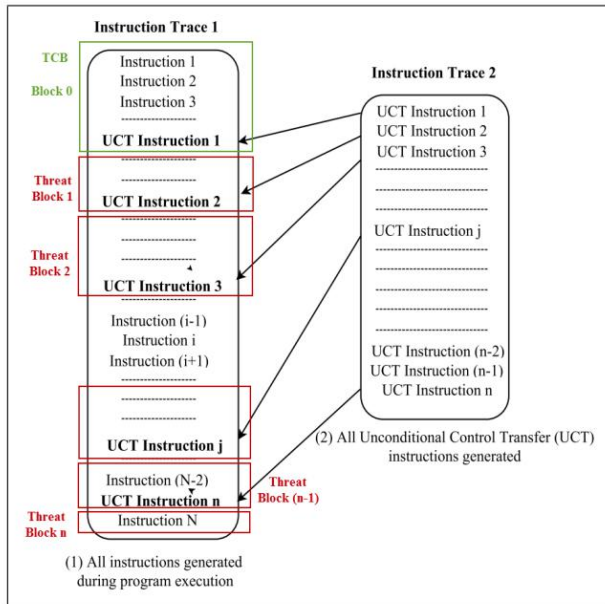


Fig. 2. Mapping of extracted unconditional control transfers to the original instruction trace

Step 4: Calculate the amount of privileged information generated ($H[PRI]$) during this activity. This can be calculated as a measure of information entropy. Assume that there exist $n1$ number of privileged instructions in the original instruction sequence, using equation (1) the information entropy generated can be calculated as follows;

Total number of instructions in the trace = N

Total number of privileged instructions = $n1$

Privileged Entropy ($H[PRI]$) generated

$$H[PRI] = - \sum n1 \cdot \log(n1/N) \quad (3)$$

Note: We observed that both N and $n1$ values will slightly change even when repeating the same test for several times. Therefore when calculating $H[PRI]$ it is required to calculate its summation.

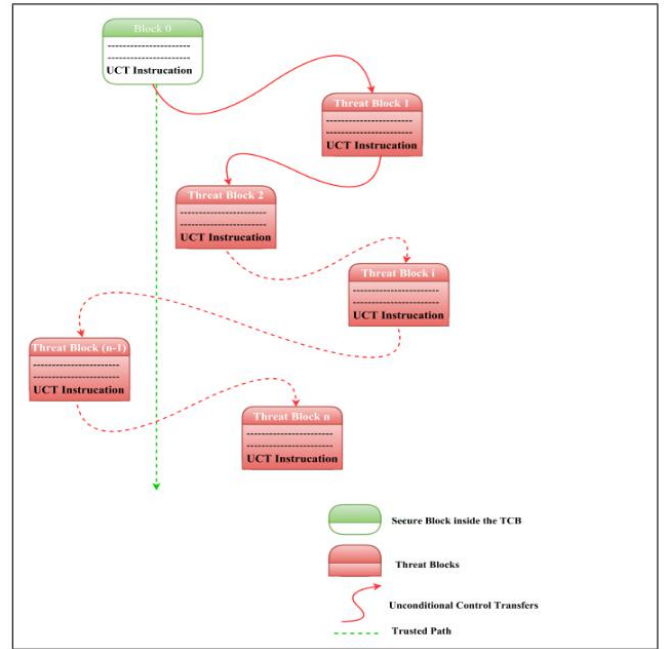


Fig. 3. The Threat Block concept

B. The Security Risk Factor (SRF)

According to equation (3), $H[PRI]$ amount of information is distributed among n threat blocks. Therefore, a given threat block will contain $H[PRI]/(n+1)$ bits of information. The test results show that in practice n to be very large and thus $(n+1)$ can be approximated as n . This is the proposed risk evaluation approximation termed Security Risk Factor (SRF).

$$RECSRF = H[PRI]/n \quad (4)$$

This is a lower bound for the risk factor for a given program execution.

V. TEST RESULTS

The proposed framework was evaluated by measuring the security strength of programs executed as sudo and root user on a virtualized

system that used Intel Virtualization Technology (Intel-VT) [1]. This hardware assisted VT extension has led to a higher volatility in microprocessor state change. The Intel-VT extensions provide 12 new Virtual Machine eXtension (VMX) instructions (VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, and INVVPID), 2 new processor modes of operation (VMX root and VMX non-root operation), and 2 VMX transition schemes (VM entries and VM exists) for transition between VMX root and VMX non-root operations [13].

The evaluation testbed was a system with an Intel Core i5 microprocessor, 8GB memory, KVM Hypervisor, Fedora 22 Host OS, and RHEL 7 guest OS. To be practically useful, RECSRF ratio should be used with a scale matching the security requirements and the sample scale used for evaluation is given in Table II. The test results are summarized in Table III and the details are in Table IV.

TABLE II
SEI RECOMMENDATION FOR CYCLOMATIC COMPLEXITY

RECSRF Ratio	Risk	Note
$X < 0.25$	LOW	Security risk difference between executions are low
$0.25 \leq X < 0.75$	MEDIUM	One execution has a moderately high security risk
$X \geq 0.75$	HIGH	One execution has a significantly high security risk

TABLE III
PROGRAM EXECUTION SECURITY STRENGTH ANALYSIS WITH RECSRF
FOR *sudo* VS. *root*

Test	Description	RECSRF Ratio	Risk
1	Modification of privileged logs	2.7354	HIGH
2	SUID execution (ping request)	1.1227	HIGH
3	Change of DNS config file	0.5300	MEDIUM
4	Network analysis with tcpdump	0.8052	HIGH
5	Modification of software firewall	0.7560	HIGH
6	Change of SELinux modes	0.7000	MEDIUM
7	Stopping the audit daemon	1.4612	HIGH

VI. CONCLUSION

This paper proposed, RECSRF, a novel concept consisting of the Runtime Execution Complexity (REC) and its evaluation method Security Risk

Factor (SRF). This is a practical scheme that can be applied without introducing any additional overhead to the system.

The novel feature of RECSRF is the quantitative evaluation of security strength of program executions. It can be used by system architects or non-technical managers to get an overall understanding about the system in terms of its runtime behaviour and interactions it makes with other program modules.

RECSRF has many advantages. It can be used to detect malicious code injects and execution redirections, to build a hardened server environment, and as an application level intrusion detection and prevention system. At system and software design stages, RECSRF can be used in determining the most secure resource combination for maximum security.

RECSRF can be used to control the growth of code bases and maintain a consistent control flow integrity. The most useful feature of RECSRF could be its usability even on production systems without performance degradation. This makes RECSRF evaluation significantly different from other conventional security evaluation methods.

TABLE IV
RECSRF TEST RESULTS DETAILS

No	Description
1	The results show the per threat block privileged information density on root execution to be nearly 250%. Therefore, the execution via root is the secure method
2	The root execution is nearly 100 % times secure than the sudo execution. Hence the root execution is recommended
3	The ratio indicates that the per threat block privileged information density on the root execution is almost twice compared to that of the sudo execution. Hence the sudo execution is recommended
4	The results show the existence of high privilege density on a root threat block. Hence the sudo execution is more secure than the root execution
5	The sudo execution is more secure than the root execution
6	The sudo execution is more secure than the root execution
7	The per-threat block privileged information density on sudo execution is higher than that of the root execution. Hence the execution via root is more secure than sudo execution

REFERENCES

- [1] D Abramson, J Jackson, S Muthrasanallur, G Neiger, G Regnier, R Sankaran, I Schoinas, R Uhlig, B Vembu, and J Wiegert. Intel virtualization technology for directed i/o. Intel Technology Journal, 10(3), 2006.
- [2] E Berlinger. An information theory based complexity measure. In Proceedings of the May 19-22, 1980, National Computer Conference, pages 773 – 779. ACM, 1980.
- [3] J M Bieman. Metric development for object-oriented software. Software Measurement. International Thompson Computer Press, London, page 75, 1996.

- [4] E Bonelli, A Compagnoni, and R Medel. Information flow analysis for a typed assembly language with polymorphic stacks. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 37–56. Springer, 2005.
- [5] N Carlini, A Barresi, M Payer, D Wagner, and T R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015. bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.
- [6] K Coogan and S Debray. Equational reasoning on x86 assembly code. In *Source Code Analysis and Manipulation (SCAM)*, 2011 11th IEEE International Working Conference on, pages 75–84, 2011.
- [7] S P Dandamudi. *Introduction to assembly language programming: from 8086 to Pentium processors*. Springer Science & Business Media, 2013.
- [8] L Davi, A Dmitrienko, M Egele, T Fischer, T Holz, R Hund, S Nürnberg, and A Sadeghi. Poster: Control-flow integrity for smartphones. In *Proceedings of the 18th ACM C on CCS*, pages 749–752, 2011.
- [9] L Davi, A Dmitrienko, M Egele, T Fischer, T Holz, R Hund, S Nürnberg, and A Sadeghi. Mocfi: A framework to mitigate control flow attacks on smartphones. In *NDSS*, volume 26, pages 27–40, 2012.
- [10] A Fog. *The microarchitecture of intel, amd and via cpus. An optimization guide for assembly programmers and compiler makers*. Copenhagen University College of Engineering, 2011.
- [11] N Francesco and L Martini. Instruction-level security analysis for information flow in stack-based assembly languages. *Information and Computation*, 205(9):1334–1370, 2007.
- [12] G K Gill and C F Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284 – 1288, 1991.
- [13] Part Guide. *Intel R 64 and ia-32 architectures software developer manual. Volume 3B: System programming Guide, Part, 2*, 2011.
- [14] P Kocher, D Genkin, D Gruss, W Haas, M Hamburg, M Lipp, S Mangard, T Prescher, M Schwarz, and Y Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.
- [15] M Lipp, M Schwarz, D Gruss, T Prescher, W Haas, A Fogh, Horn, S Mangard, P Kocher, D Genkin, Y Yarom, and M Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [16] T J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.
- [17] A Minai and D Braha. *Complex engineered systems: Science meets technology*. 2006.
- [18] G Neiger, A Santoni, F Leung, D Rodgers, and R Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.
- [19] N L Petroni Jr and M Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on CCS*, pages 103–115, 2007.
- [20] J Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Conference*, pages 294–311. Citeseer, 1984.
- [21] A Sabelfeld. Language-based information security. In *Foundations of Computer Security*, page 99, 2003.
- [22] C E Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.
- [23] A Smirnov and T Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS*, 2005.
- [24] D Xu and K E Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, (4):265–278, 2006.



Asanga Wanniarachchi is a Security Researcher, attached to the Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka, and currently working on reverse engineering techniques and run-time analysis of disassembled machine instructions for developing security indexing tools for quantitative risk assessment. He has completed his Masters Degree in Computer Science, specializing in Security Engineering from University of Moratuwa Sri Lanka, and received the highest grade for Masters Degree Research. Asanga is a Certified Information Systems Security Professional (CISSP), and holding several other industry certifications from Cisco Systems Inc, Red Hat Inc and Oracle Corporation, and a Professional Member of the Computer Society of Sri Lanka.



Dr Chandana Gamage is a Senior Lecturer in the Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka, and was the former Head of the Department. After receiving his PhD from Monash University in Melbourne, Australia specializing in Computer Network Security and Cryptography, he did his post-doctoral work with Prof Andrew Tannenbaum at Vrije Universiteit in Amsterdam, Netherlands. The active research areas of Dr. Gamage are in the fields of Cryptography, Computer and Network Security, Secure Virtualization, Virtual Reality, Augmented Reality, Mixed Reality, Embedded Systems, and Enterprise Systems Architecture. Dr Gamage is Chartered Engineer and a Commissioned Officer in Sri Lanka Army.