LANGUAGE-BASED SECURITY TO APPLY COMPUTER SECURITY

Ali Ahmadian Ramaki[1] and Reza Ebrahimi Atani[2]

[1]Department of Computer Engineering, Guilan University, Rasht, Iran
ahmadianrali@msc.guilan.ac.ir
[2]Department of Computer Engineering, Guilan University, Rasht, Iran
rebrahimi@guilan.ac.ir

ABSTRACT

Language-based security is a mechanism for analysis and rewriting applications toward guaranteeing security policies. By use of such mechanism issues like access control by employing a computing base would run correctly. Most of security problems in software applications were previously handled by this component due to low space of operating system kernel and complicacy. These days this task by virtue of increasing space of OS applications and their natural complicacy is fulfilled by novel proposed mechanisms which one of them is treated as security establishment or using programming languages techniques to apply security policies on a specific application. Language-based security includes subdivisions such as In-lined Reference Monitor, Certifying Compiler and improvements to Type which would be described individually later.

KEYWORDS

Security, Security Policy, Programming Languages, Language-Based Security

1. INTRODUCTION

Growing use of the Internet, security of mobile codes is one of important challenges and issues in today's computational researches. On increasing our dependency on large global networks such as the Internet and receiving their services in order to perform personal routines and spread global information over these global networks and even download from this perilous area is potentially susceptible to destructive attacks from attackers and may be followed by irrecoverable effects. We do not still forget pernicious attacks such as "Mellisa" and "Happy 99" and while downloading plug-ins in the internet packages careful attention is needed and how exhaustive outcomes they have caused. Recent researches show these types of security issues are on the rise.

New studies reveal vital foundations such as transportation, communication, financial markets, energy distribution, health and etc. fully rely on a computing basis which can hardly be a bare justification. We have hazardously depended on big software systems which their behaviours do not totally penetrate and most of failures occur unpredictably. Thus it is a necessity as to how to utilize the internet applications through safe and secure. Today with respect to expansion of computational environments, safety topic in term of mobile codes is indispensible. For instance, having downloaded an application from the internet from an unknown source how could we warrant it would not carry an unwanted file which may put system safety as risk?

A way for understanding of the situation is use of language-based security. Throughout the method, security information of an application programmed in a high-level language is extracted during compilation of the application that is a compiled object. The extra security information includes formal proof, notes about the type or other affirmable documents. The compiled object is likely to be created alongside destination code and before running the main code is automatically examined to warn of errors types or unauthorized acts. “Java ByeCode” affirmative is an example for the issue in question. The chief challenge is as to how to create such mechanisms such that in the first place they have the desirable performance and in the second place they are not revealing to others as much as possible [1].

Following the paper, we declare motivation for language-based security methods toward guaranteeing security in section 2. The issue literature is reviewed in section 3. Traditional actions to affirm the security in computer systems are investigated in section 4. In section 5 language-based security framework and later desirable techniques are explained in section 6 and a case study on safe and unsafe language are presented in section 7. Finally in section 8 the conclusion is drawn.

2. MOTIVATION

Computer security is usually practicable by operating systems at a comprehensive level. Despite of growth of operating systems both in size and in complicacy, applying computer security policies is drastically difficult. For this reason, new attacks likely occur against operating systems security mechanisms which may be followed by successes. To speak more accurately, computer systems fulfil computer security mechanisms at a lower level such as access control or maintenance of files while most of beneficial practices are all high-level or at a specific level of convention such as pernicious worms breaking into computer systems by email applications. The keynote, in here is to offer a specific convention for each dangerous attack and practice then to guarantee security. Thus operating systems kernels only confirm coarse-grained policies.

Such assurance is considered ad hoc and since applications are programmed in specific languages, guaranteeing security for specific applications with the help of programming languages is a predominant topic on which many researchers have performed and are known as language-based security.

The significant advantage of language-based security toward security assurance of applications is that security policies and execution mechanisms are performed by expanded techniques in programming languages and this situation is created by natural capabilities of programming languages. Security assurance in computer systems with respect to language-based techniques and advances in programming languages has drastically developed [7]. Language-based principal framework is shown in figure 1.

3. A REVIEW ON ISSUE LITERATURE

3.1. Two Principles in Computer Security

To understand language-based security more accurately we need to introduce two principles in computer security systems and provide them with detail descriptions.

- **Principle of Least Privilege (PoLP)**: while running accomplishment policies, each principle is supposed to have least possible access to be applied.

- **Minimal Trusted Computing Based (MTCB)**: components which should operate properly to confirm execution system properties, such as operating system kernel and hardware. That is mechanism in use fulfills big tasks while small. Smaller and simpler systems have less errors and improper interactions which is quite appropriate to install safety.

To guarantee security policies two principles PoLP and MTCB are still valid, however some new mechanisms and safety policies are required. This is an impetus toward designating update safety policies and optimal execution mechanisms on action systems basis.

3.2. Necessary Words

To understand, some of the most important concepts are listed below:

Reference Monitor: references monitor how programs run in the objective system and if the system sidesteps a security policy, it would prevent it from proceeding. Security mechanisms may be installed on system hardware and system software usually either implements reference monitors directly or facilitates their implementation. For instance operating system as a software agent implements a reference monitor in order to make files accessible while switching context or trap occurrence, operating system causes evoking system commands by control transmission. To do so operating system or reference monitor should remain safe against interruption. Thus reference monitor program must be kept in a memory section to be guarded from hardware point of view.

Safety Policy: confirms an application runs at a desirable safety level and adopts different meanings in different areas. A safe status definition for an application, safety policy is specific purpose for that application for example we need vital system data never to be overwritten, it is in here a failure occurs and control software should signal a message.

Generally speaking every safety policies must guarantee preliminary safety properties for an uncertain machine code running locally on a system:

- **Flow Control Safety**: all branches, jumps and evocations to a random location should within address space of program instruction. All evocation should point to valid functions within program instruction section. Moreover, all return addresses from procedures and subprograms must remain within program code space as well.

- **Memory Safety**: all program accesses to data memory stay within memory section, heap memory and stack space earmarked to that specific program.

- **Stack Safety**: for architectures utilizing stacks in order to store return addresses from subprograms and procedures while running programs it is a crucial issue that at the top section of stack which these addresses are kept minor errors occur because of possibility of address deletion.

Another problem forming on the side is known as Type Safety or less famous Typing Discipline which in this method raw data and codes are allocated destinations. Return type of a regular function is assessed by destinations of its input variables. Assuming we have a function accepting integer values. If the output is as below while returning from function and with registers and holding integer values the output is certainly integer as shown in (1).

$$r_1\text{: int} * r_2\text{: int} \longrightarrow r_3\text{: int} \qquad (1)$$

Trust: safety shapes up in terms of trust. There are, de facto, various levels for trust which we break them down in two classes for simplicity; those groups that are trusted and those that are not so such that they are infusible regarding trust boundaries. All trusted software is composed of reliable basic codes. Every software safety mechanisms hinge on some fail-safe codes.

Performance vs. Safety: it is pleasurable to have severe safety warranty and decent performance although they almost conflict. Exerting safety mechanisms coming in delay is a burden against fast-execution of programs which increases eventually execution time that is drop of performance and the goal is establishing a compromise between the two points. Language-based security techniques as we would describe later make efforts to improve both aspects.

4. TRADITIONAL APPROACHES TO APPLY SECURITY

Traditional methods to safety issue within computer systems include:

- Utilizing OS kernel as a reference monitor
- Cryptography
- Code instrumentation
- Trusted compilation

These mechanisms offer a constant amount of preliminary security policies benefitting from low flexibility. In future we scrutinize them in detail.

1. **Utilizing Operating System Kernel as Reference Monitor**: this method is the oldest but the most exhaustive mechanism in use to guarantee security policies in software systems and fulfils single actions on data and critical components of system through operating system kernel. Kernel is an indispensible component of operating system code retrieving vital components and data directly. The rest of programs are somehow constrained in order to access these data and components such that kernel plays a role of proxy interchanging messages for communication. In general, not only does kernel foil suspicious codes execution in order to avoid a probable breakdown but supervises all accesses and safety policies accomplishment. One of disadvantages of the approach may originate from high-overload of context switch between various processes as they are supposed to receive certification from kernel to retrieve their demanded data and components.

2. **Cryptography**: by this method makes it possible to install safety at a sensible data transmission level in an unreliable network and make use of a receiver as a verifier. Power of cryptography methods is as much complex as hypotheses. Digital Encryption Standards (DESs) are susceptible to violation by a sufficient amount of damaging codes. Cryptography thus cannot guarantee downloaded codes from a network to be safe. It is only able to provide a safe transmittal space for these codes through the Internet to avoid intrusions and suspicious interference.

3. **Code Instrumentation**: Another approach practiced by operating system in some systems to inspect safety level of a program from various aspects such as writing, reading and programming jumps. Code instrumentation is a process through which machine code of an executed program is changed and main action consequently could be overseen during execution. Such changes occur in sequence of program machine code for two reasons; first, behaviours of changed code and initial code equal. It suggests that initial code did not violate safety policy and second, if violation by initial code occurs, changed

code is immediately able to handle the situation by two options; either it recognizes violation, gains control from system and terminates destructive process or prevents fatal effects which are likely to affect the system soon.

For instance let's suppose a program needs to be run on a machine with certain hardware specifications. To do so let's assume the program is loaded within a continuous space of memory addresses [$c2^k$, $c2^k + 2^k - 1$] where c and k are integer numbers. The program then links to run and after execution and obtaining destination code, by altering values of and jumping to another address space of memory for indirect addresses, the code in question is ready to run.

Of conditions of a safe program is that after modifying direct addresses to indirect ones the main action of new destination code does not permitted to change and must follow the previous objectives. This drops under Software Faults Isolation (SFI). According to the SFI, software components of a program remain within the same address spaces of hardware. To guarantee this, a software reference monitor in order to individualize components to logical address spaces is exerted. By this reference it is finally assured every reading, writing and jumping take place within the same logical address space, however its setback is due to high overload of checking which may compromise the communication rate between components. Recent studies prove the improvements of final performance by a combination of code instrumentation and language-based security approaches [4], [6].

4. **Trusted compiler**: this method is fulfilled by a component known as trusted compiler. By making virtue of codes limited access, compiler attempts to generate a code which is trusted. There are two alternatives for operating system kernel to warrant reliability of compiler.

   If compiler is an independent component compiler must comply with a rule to ensure kernel that the generated code is produced by the same trusted compiler. To do so, the compiler somehow needs to mark the code and it is a type of signature. Having checked the signature kernel makes sure of code correctness. The other alternative happens when compiler is built into the kernel. But a drawback to the second approach is that when the size of generated code by trusted compiler is not small enough it then takes up the space of kernel. Hence according to traditional approach a trusted compiler fitted into the kernel is applied, although the method carries its certain burdens.

5. LANGUAGE-BASED SECURITY

As mentioned before, the Internet is a susceptible area for destructive agents to penetrate into computer systems influencing the security considerably. For instance the worm "Morris" broadcasted on October in 1988 was an intruding worm through the Internet which affected 5 percent to 10 percent of 6000 machines connected to the network. "Morris" employed some ulterior methods to access a host machine. "Love Bug' and "Mellisa" were two famous viruses utilizing the email services to be propagated. The reason why these viruses succeeded was that they were unintentionally authenticated by the user in order to have desirable privileges for their destructive codes. Plus, operating system kernel was totally unaware of proceedings.

Of preventive counteractions is to scan the computer system for viruses or suspicious codes, not complete yet. One of important approaches on which we focus is to utilize code semantic and behaviour in order to detect infections and is known as language-based security.
Times possible to prevent bad events include:

- Before execution: code analysis, code rewriting, creating backup from user's data
- During execution: execution with monitor and stopping malfunctions; detection of an abnormality occurrence and counteraction, detection of malfunction occurrence and recording of events.
- After execution: evocation of monitor and scan of Log. Schneider typically explains language-based security as: "a set of techniques founded on theory and programming languages such as definitions, types and optimization that are able to provide answers to safety problems." By such definition approaches like SFI and Security Automata SFI Implementation (SASI) are treated as examples for language-based security.

In computer systems, compiler usually interprets a program in a high-level language. Assembler of destination machine then issues Hex code of the program to the hardware to let it start. Compiler obtains information about programs while compiling them. The information includes variables values, types or specified information and may be analyzed and modified in order to optimize produced destination code by compiler.

After successful compilation, extra information are mostly rallied which can provide information about security of destination compiled code. For example in case the program is written in a safe language before compilation filter of type check must be complete successfully. So codes about security information should also be generated alongside destination code in order to run on the hardware during compilation process. This information as a certificate is created before program execution and it starts running before produced destination code execution to ensure security policies of the specific convection is met. Such process is already shown in figure 1.

As explained, some of this extra information is generated during interpretation of a program about security aspects of destination code. For instance when a program is written in a Type Safe Language, after initiative examinations by compiler the program is inspected in term of type and compiler hence guarantees safety of instruction memory. If code consumer does not access such extra safety information his decision so as to securely run downloaded application from an unknown source is easier to make.

Concept of language-based security is given to such extra information extracted from a program written in a high-level language and while compiling this extra information package also called certificate. During downloading applications from the Internet or any other unsafe tool, this package of extra information is uploaded as well. Code consumer is able to evoke a verifier program before running an application to confirm the certificate and code then run it.

One of consequential objectives of language-based security is that responsibility of verifying the code of a program from user's side is removed and transmitted to the code provider. It means whenever a code provider intends to upload an application is supposed to supply a certain certificate about safety aspects of code execution. So, the responsibility of code user to prove if the code is safe is narrowed down to check safety of the code.

Code providers take advantage from various techniques to produce such certificate. Some of the most important ones are:

1. **Proof Carrying Code (PCP)**: produced certificate by the code provider is a first order logic proof wherein a set of safe conditions to run code is supplied and user checks their correctness on the downloaded application while running the code.

2. **Type Assembly Language (TAL)**: certificate is a type reminder such that verifying process on the user's side inspects code structure in term of type.

3. **Efficient Code Certification (ECC)**: in this approach contains extra information about destination code checking concept structures and code objectives according to type theory information.

6. LANGUAGE-BASED SECURITY TECHNIQUES

A reference monitor is a program execution and prevents the program if it violates the safety policies. Typical examples for reference monitor are operating systems (hardware monitor), interpreters (software monitor) and firewalls. Most of safety mechanisms, today, employ reference monitor. It should:

- Have accessibility to the information about what the program fulfills.
- Stop the program once it encounters violation.
- Save the code and program status from destructive interferences.
- Have low overload.

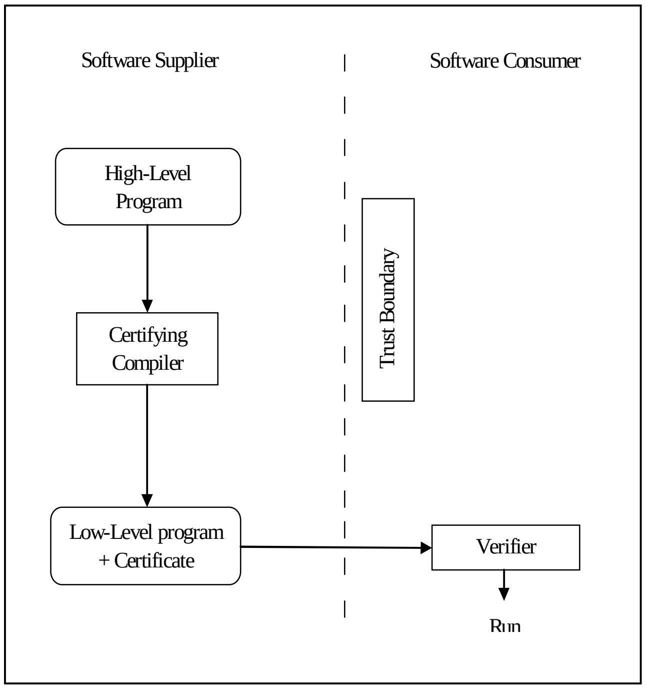

Figure 1: Overview of Language-Based Security

In future we define a security policy and what type of safety policies are guaranteed by reference monitors. Some of safety policies are warranted by reference monitors called safety properties. Some of them are approximately irresolvable by reference monitors and do not accept the program if it breaks the policies.

The hypothesis in reference monitor issue is that a reference monitor can access all computation conditions. The reference monitor possesses indefinite states but it cannot predict future i.e. it cannot forecast which program to terminate. Reference monitor only observes a sequence of programs. Let's assume we guarantee safety policy P by reference monitor wherein P represents a unique sequence which is brought in equation (2).

$$P(s) = \forall\, \sigma \in s,\ P(\sigma) \qquad (2)$$

A set of executive sequence is a safety property if every member is individually determined by the sequence and does not depend on other members. In some cases P must be continuous. On this condition, some sections may not be established because monitor decision is made within a limited time and we consequently follow through sequences execution. According to equation (3) we see:

$$\square\sigma, P(\sigma) \Leftrightarrow (\square i, P(\sigma\square[...i])) \tag{3}$$

A prediction of P on a set of sequences simultaneously meeting conditions 2 and 3 is a safety property. The rule is well-known as Schneider guaranteeing no malignant event would happen. We can say, as a result, that a reference monitor does not affirm a safety policy which is not a safety property. It proves, de facto, reference monitor is capable of guaranteeing every safety policies.

Reference monitors are traditionally based on hardware and guaranteed by operating system and its kernel. Such reference monitors were able to warrant safety properties of files manipulated by system programs and these manipulations contain reading, writing, execution or adjustments to list of files accessibility control; that is what changes each user is permitted to apply on each file. Generally speaking, high-level objectives that reference monitor is able to satisfy is files and sources accessibility and fullness.

Applying safety policies by reference monitor inside operating system might be useful for old systems of which operating systems benefitted from low space and had a limited amount of communication rate and contexts switch to have the program run. But because of pass of time, operating systems complicacy and increase in their codes space in 1980 approaches were founded in order to have authentication for managing such monitors off operating systems kernel environment. On this condition, tasks like context switch codes, Transmission Lookaside Buffer (TLB), trap management, interruptions and access to peripheral devices were dealt by kernel and other tasks were considered as processes to run; actions such as system file, communication protocols, paging algorithms. The reason is to augment flexibility and safety level.

Building blocks of language-based security are the program rewriting and analysis. By rewriting it is guaranteed that programs do not perform unauthorized behaviours through a program analysis in order not to let programs break the policies. In future we describe language-based security mechanisms individually. These mechanisms are divided in two categories; first, rewriting phase formed by in-lined reference monitor mechanisms and second, analysis phase warranted by type safe programming. Later we expound another novel approach famous as certifying compilers.

1. **In-lined Reference Monitor (IRM)**: a mechanism fulfilled by operating system in traditional approaches to supervise programs flawless execution and confirmation of objective safety policies is that reference monitor and objective system are located in distinct address space. Alternative approach is an in-lined reference monitor; a similar task which is performed by SFI. This component fulfills safety policy for objective system by stopping reading, writing and jumps in the memory outside a predefined area [3]. One of methods thus, is the merge of the reference monitor with objective application. In-lined reference monitor is specified by definitions below:

    - **Security events**: action to be performed by reference monitor.
    - **Security status**: information to be stored during a safety event occurrence according to which a permission to progress is issued.

- **Security updating**: sections of the program running in response to safety events and updating safety status.

While loading the program, program rewriter IRM generates a verified application having its fullness and safety confirmed in which there is no safety policy to be broken. SASI is the first generation of IRM proved by researches to be an approach guaranteeing policies in question. The first generation is programmed in Assembly80x86 and the second generation is programmed in Java [2]. SASI x86 that is compatible with Assembly 80x86 is the graphical output of gcc compiler. The destination code generated meets the two conditions below:

- The program behavior never changes by adding NOPs.
- Variables and addresses of target branch marked with some tags by gcc compiler are matched during compilation.

So the first version is comprehensively employed in order to save the program memory data. In the second version of IRM, JVML SASI, the programmed is preserved in term of type safety. JVML instructions provide information about the program classes, instances, methods, threads and types. Such information can be utilized by JVML SASI to supply safety policies in applications [5]. Rewriting components in IRM mechanism generate a verifying code with related destination code by this extra information [10].

2. **Type System**: the main objective is to prevent error occurrence during the execution. Such errors are identified by a type checker. The importance of this case is that a high-level program certainly does have many variables. If these variables of a programming language are within a specific area we technically say the language is a type safe. Let's assume variable x in Java is defined as a Boolean and whenever it is initiated False the result is !X(not x) that is True. If variables are under a condition such that their values are within an undefined area we say the language is not type safe. In such languages we do not meet types but a global type including all possible types. An action is fulfilled by arguments and output may contain an optional constant, an error, an exception or an uncertain effect [8].

   Type system is a component of type safe languages holding all types of variables and type of all expressions are computed during execution. Type systems are employed in order to decide a program is well-formed. Type safe languages are explicitly known as typical if types are parts of syntax otherwise implicit type.

   Type safe languages such as ML and Java guarantee program actions to be applied only on proper values. They also warrant some safety properties such as memory and control security. Type systems supporting type abstract permit programmers to define new types and consequently fulfill actions by means of them. If an unauthorized code tends to perform inappropriate actions on improper types it would be thwarted by type checker. The keynote idea lies in here that checking during program execution on the user's side must be triggered in generator section while programming. Recent advances in designing type systems make definition of useful safety properties possible and facilitate programs execution on the user's side by drop of type checking while programming. Such mechanism gives rise to memory safety policies accomplishment and flow control.

3. **Certifying Compiler**: the main blind drawback of type-based approaches to safety establishment is that they suppose some hypotheses about high-level language. The program must be written in a high-level language which is well-behaved in terms of type

and action concept. Moreover, the programmer is supposed to write codes in a high-level language and end user should properly implement the program from type and action concept aspect which causes lower flexibility. A Certifying compiler is a compiler that the data given to it guarantees a safety policy, generates a certificate as well as destination code which is checkable by machine i.e. it checks policies in question [9].

For instance, a company generates a Java code while running programs which some extra information about code safety to the form of Java Byte Code is produced. Certifying compilers are the most significant tool to run safe programs since they follow through without considering computation bases. To examine if the output code of a certifying compiler commit some policies, an automatic certificate checker is employed. Such checker examines the output of certifying compiler if the destination code holds properties included in the certificate. For example, JVML examines such code during execution a Java program and before execution of main code to check if the program is type safe. An instance of such compilers development is PCC.

7. Case Study on Safe and Unsafe Language

As mentioned, a language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data." So, we can deduce that C++ is not type-safe according to this definition at least because a developer may cast an instance of some class to another and overwrite the instance's data using the "illegal" cast and its unintended methods and operators.

Java and C# were designed to be type-safe. An illegal cast will be caught at compile time if it can be shown that the cast is illegal; or an exception will be thrown at runtime if the object cannot be cast to the new type. Type safety is therefore important because it not only forces a developer to write more correct code, but also helps a system become more secure from unscrupulous individuals. However, some, including Saraswat, have shown that not even Java is completely type-safe in his abstract.

Java comprises a language-based mechanisms designed to protect against malicious applets. The Java runtime environment contains a byte code verifier that is supported to ensure the basic properties of memory, control flow, and type safety. There is also a trusted security manager that enforces higher-level safety policies such as restricted disk I/O [7].

8. Conclusions

Security in computer systems holds an importance stand. In traditional approaches computer systems safety is founded on two principles of minimal access privilege and computing base. In such approaches the safety is warranted by operating systems and kernels which the kernel acts as a proxy for other processes running on the system. Because of technology advances, complicacy of operating systems in terms of tasks and increase in kernel codes for supporting properties such as graphic cards and distributed file system, new approaches install safety which are proved to be high performance like safety establishment by using programming techniques. Such techniques drop under three main categories: in-lined reference monitor, type system and certifying compilers which are described separately.

REFERENCES

[1] J.O. Blech, A. Poetzsch-Heffter, "A Certifying Code Generation Phase", Proceedings of the Workshop on Compiler Optimization meets Compiler Verification, pp. 65-82, 2007.

[2] U. Erlingsson, F.B. Schneider, "IRM enforcement of java stack inspection", In IEEE Symposium on Security and Privacy, Oakland, California, 2000.

[3] R. Wahbe, S. Lucco, T. Anderson, S. Graham, "E□cient Software-Based Fault Isolation", In Proc.14th ACM Symp. on Operating System Principles (SOSP), pp.203-216, 1993.

[4] K. Crary, D. Walker, G. Morrisett, "Typed Memory Management in a Calculus of Capabilities", In Proc. 26th Symp. Principles of Programming Languages, pp. 262-275, 1999.

[5] U. Erlingsson, F.B. Schneider, "SASI Enforcement of Security Policies: A Retrospective", 1999.

[6] F.B. Schneider, G. Morrisett, R. Harper, "A Language-Based Approach to Security", Lecture Notes in Computer Science, pp. 86-101, 2001.

[7] D. Kozen, "Language-Based Security", Mathematical Foundations of Computer Science, pp. 284-298, 1999.

[8] R.Hahnle, J.Pan, P. Rummer, D. Walter, "Integration of a Security Type System into a Program Logic", Theoretical Computer Science, pp. 172-189, 2008.

[9] C. Yiyun, L. Ge, H. Baojian, L. Zhaopeng, C. Liu, "Design of a Certifying Compiler Supporting Proof of Program Safety", Theoretical Aspects of Software Engineering, IEEE, pp. 127-138,2007.

[10] M. Jones, K.W. Hamlen, "Enforcing IRM Security Policies: Two Case Studies", Intelligence and Security Informatics, IEEE, pp. 214-216, 2009.

Authors

Ali Ahmadian Ramaki was born in Iran on August 10, 1989. He recieved his BSc degree from university of Guilan, Iran in 2011. He is now MSc student at university of Guilan, Iran. His research interests in computer security, network security and intelligent intrusion detection.

Reza Ebrahimi Atani received his BSc degree from university of Guilan, Rasht, Iran in 2002. He also recieved MSc and PhD degrees all from Iran University of Science and Technology, Tehran, Iran in 2004 and 2010 respectively. Currently, he is the faculty member and assistant professor at faculty of engineering, University of Guilan. His research interests in cryptography, computer security, network security, information hiding and VLSI design.