International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:10, No:4, 2016

Four Phase Methodology for Developing Secure Software

Carlos Gonzalez-Flores, Ernesto Liñan-García

***Abstract*—**A simple and robust approach for developing secure software. A Four Phase methodology consists in developing the non-secure software in phase one, and for the next three phases, one phase for each of the secure developing types (i.e. self-protected software, secure code transformation, and the secure shield). Our methodology requires first the determination and understanding of the type of security level needed for the software. The methodology proposes the use of several teams to accomplish this task. One Software Engineering Developing Team, a Compiler Team, a Specification and Requirements Testing Team, and for each of the secure software developing types: three teams of Secure Software Developing, three teams of Code Breakers, and three teams of Intrusion Analysis. These teams will interact among each other and make decisions to provide a secure software code protected against a required level of intruder.

***Keywords*—**Secure Software, Four Phase Methodology, Software Engineering, Code Breakers, Intrusion Analysis.

I. Introduction

THIS research is the natural continuation of our previous work [1]. Our previous work dealt only with the technique of obfuscation for the development of secure software. On that work we demonstrate analytically that our basic methodology (for the specific case of using only obfuscation as a means to secure the software) is better that not using a methodology at all. This current work encompasses all current techniques used for secure software development including self-protected software.

Secure software is defined as the software that is designed and developed with the idea of making it impossible/hard/very hard for an intruder to reverse engineer the algorithms, procedures and methodology of the underlying system [2]-[4].

The amount of effort that one has to put into securing the developing software should be related to the type of intruder one wants be protected from.

We are assuming that all intruders have only access to the machine code (binary or assembly code). The security of the high level code and software documentation is not considered here; since they should be part of external security mechanisms (i.e. the high-level code is leveled as confidential or secret by the developing organization, and protected as such).

Another assumption made in this methodology is that the requirements/specifications can each be identified as being security-related or not.

Carlos González and Ernesto Liñan are with the Facultad de Sistemas, Universidad Autónoma de Coahuila, Arteaga México (e-mail: c.gonzalez.flores@uadec.edu.mx, ernesto_linan_garcia@uadec.edu.mx).

II. Methodology

A. Secure Software Development

We will analyze three ways of doing the development of secure software:

- Have the secure software embedded with the application. Examples of this technique are:
  - User Detection [4]
  - Internal Anti-Tampering [2],[3]
  - Anti-debugging [7]
  - Internet usage [4]
- With transformations done to the applications software. Examples of this technique are:
  - Obfuscation at the high level code [8]-[12]
  - Obfuscation at the machine level code [13]
- With a shell controlling the ins and outs of the application. Examples of this technique are:
  - Runtime Application Self-Protection (RASP) [14

B. Phases

The methodology proposed in this paper consists of four phases:

- Phase-1. This phase is the software development and testing of all the non-security related software.
- Phase-2. In this phase, the self-protected secure software is developed, tested and incorporated into the non-security related software produced in Phase-1.
- Phase-3. This phase will use transformation to the software generated in Phase-2. The transformations are done at high-level code and at machine-level code. All the transformed code will be tested to comply with the requirements of the software.
- Phase-4. This phase will generate and test all the shield code used in the software.

C. Intruders

We define four types of intruders:

- Level-1: A casual attacker. The attacker has the software and he/she is not technically knowledgeable to retrieve data or algorithms from the machine code software. A minimum level of security is needed.
- Level-2: A hacker attack. This attacker has the knowledge to retrieve data or algorithms from the machine code sources of the software. Attacks of this kind need to have security procedures used for the development of code. This intruder may or may not have initial economic gains plans

for the intrusion. In most cases it is the intellectual challenge that motivates this intruder (i.e. hacker), but economic gains may not be very far behind.

- Level-3: An institution attack. This attack is done by an institution with all the resources of such an institution. The most common cases are industrial espionage. The economic gains are the main reason for the intrusion. In most cases with enough time and money, any secure software may be cracked. Therefore, the developing team should always work with the goal of making the intruder's effort needed to break the secure code high enough for them not to be cost effective.
- Level-4: A government attack. This attack is done by a government agency with all the resources (technical and legal) available for such agency. Since in most cases with enough time and money any secure software may be cracked, it is recommended that techniques for intruder detection with the respective actions to take (covert and non-covert) be included in the secure code [5], [6]. This level of protection requires the use of the most sophisticated security algorithms.

Most current approaches/methodologies suggest that security is not a feature that can be added to extend the functionality of software, but is an essential building block and key architectural design characteristic of reliable software [15]-[17]. We partially agree with these approaches methodologies, the problem starts on the operational part of the development. As it has been pointed out [1], for example, that it is difficult to develop straight obfuscated software and follow Software Engineering practices. Also, because of the nature of self-protected software it is easier to have the non-secure software done first and then add the self-protected features to this software, rather than doing the development at the same time. What we propose is to have a global approach in the design of secure software, but use several phases to develop such software.

D. Teams

What we are proposing here is the division of work between several teams of software engineers.

1. The Developing Teams

- We will have the Software Engineering Developing team (SED), developing the software following all the guidelines and best practices described in Software Engineering [18]-[22].
- A Self-Protected Software Developing Team (SSP) of secure self-protected high level code experts, which will use secure self-protected software techniques on the high level code (techniques like User detection, Runtime Application Self-Protection-RASP, etc.) [8]-[12], and [23]-[28] in the code produced by the SED.
- A Self-Protected Machine Code Developing Team (SPM) of secure machine code experts, which will apply secure techniques to the compiled product generated by the SSP [13].
- A Transformation High Level Developing Team of transformation secure high level code experts (THL), which will use secure software techniques on the high level code (techniques like obfuscation, anti-reverse engineering, etc.) [8]-[12], and [23]-[28] in the code produced by the SED.
- A Transformation Machine Code Developing Team (TMC) of transformation machine code experts, which will apply transformation techniques to the compiled product generated by the SSP [13].
- A Secure Shield Code Developing Team of secure shield code experts (SSC), which will use secure shield software techniques on the machine code generated by the THL team [14].

2. The Compiler Team (Could Be One or Many)

- A Compiler Team (CT). This team is usually included as part of the software developing team. But in our case, we have two sets of people involved in developing software, the software engineers developing the system software, and the High Level Secure Software Team developing the secure software to be part of the developed system software. Therefore, we name the CT in both instances to make sure the process flow in our methodology is clearly understood.

3. The Specification and Requirements Testing Team

- A Specification and Requirements Testing Team (SRT). This team will have the responsibility of designing and applying the tests for the developed software. The testing will have to satisfy all the specifications and requirements of the system. In the first two stages of the methodology this team will test only the non-related security requirements/specifications, but at the final stage the testing will have to satisfy all the specifications and requirements of the system.

4. The Code Breakers Teams

Note that for all the code breakers team: The team should be made of one-two person teams for level-2 threats and the whole Code Breakers Team for level-3 and level-4 threats,

- A Self Protected Code Breakers Team (SPCB), which will try to break the self-protected secure code generated by the SSP and the SMT teams. This team should be made of self-protected software security experts that know software development, rather than experts in software development with knowledge of self-protected software security.
- A Transformation High Level Code Breakers Team (THCB), which will try to break the transformed secure code generated by the SSP and the SMT teams. This team should be made of transformation software security experts that know software development, rather than experts in software development with knowledge of transformation software security.
- A Transformation Machine Code Breakers Team (TMCB), which will try to break the transformed secure code generated by the SSP and the SMT teams. This team should be made of transformation software security experts that know software development, rather than experts in

software development with knowledge of transformation software security.

- A Shield Code Breakers Team (SCB), which will try to break the secure shield code generated by the transformation developing teams, and the SMT teams. This team should be made of secure shield software experts that know software development, rather than experts in software development with knowledge secure shield software.

5. The Intrusion Analysis Teams

- A Self Protect Intrusion Analysis Team (SPIA), which will analyze the results of the SSP and pass the results and new secure proposals for the SED and SSP teams. The members of this team should be the elite of the self-protect software security personnel. They not only need to be experts in the field of self-protect software security, but also need to be innovators of new algorithms or paradigms to solve self-protect software security problems.
- A Transformation Intrusion Analysis Team (TIA), which will analyze the results of the THL or the TMC and pass the results and new secure proposals back for the THL and TMC teams. The members of this team should be the elite of the transformation software security personnel. They not only need to be experts in the field of transformation software security, but also need to be innovators of new algorithms or paradigms to solve transformation software security problems.
- A Shield Intrusion Analysis Team (SIA), which will analyze the results of the SSC and pass the results and new secure proposals for the SSC team. The members of this team should be the elite of the shield software security personnel. They not only need to be experts in the field of shield software security, but also need to be innovators of new algorithms or paradigms to solve shield software security problems.

III. Process Flow

Figs. 1-3 describe the proposed methodology process.

Once a piece of software is liberated by the SED team (and compiled), it goes to the SRT to test all the non-security related requirements and specifications of the project. If it fails, the results go back to the SED to correct the problems with the failed requirements/specifications. If the SRT passes all the tests, then the high level code is passed to the SSP team to start work on securing it.

It should be clear that once the code is passed to the SSP, any changes/modifications to the SED generated source code implies that the cycle of re-compiling and re-testing has to be repeated before this new software is passed to the SSP. If this cycle is repeated very frequently by any piece of software, the cost of development could be highly increased.

Once the securing is done by the SSP, the software is compiled. This software has to be tested again to check that modifications done by the SSP process did not break any of the non-related requirements/specifications test. Thus, the secure code generated by the SSP and complied is passed to the SRT to again re-test all the requirements/specifications. It is expected that the SRT non-related tests are the same used in all cases.

If the SRT tests fail, the results go back to the SSP team to modify the necessary code to pass the failed requirements/ specifications. If on the other hand all the SRT tests are passed, then the code is ready to be tested by the Self-Protect Code Breaker Team. It is important to point out that even though there were no tests done for security related requirements/specifications at this point, the SSP team was expected to address all the pertinent security related requirements/specifications in their development. The final testing of all security related requirements/specifications will be performed in phase-4 after the Secure Shield Code Developing Team (SSC) is done.

The Code Breaker Team will try to act as the maximum intruder level that this software is trying to protect. Since we still have several levels of protection to use in the methodology, then, we recommend that the all the CBTs consider the code passed the test if the security of protection is adequate to protect against one level lower than the expected level of intruder, except for the last CBT which is the Shield Code Breakers Team and all the security requirements should be met by then. If the code passes the CBT tests, then the code is ready for the next level of security. If the CBT determines that the level of protection is not adequate, then this information is passed to the appropriate Intruder Analysis Team to analyze the results and report what the problems are, and if possible recommend solutions including new security procedures. This information is passed to the appropriate security development team (SSP, SPM, THL TMS, OR SSC), and a new cycle begins.

The code will not go into the next step until the appropriate SRT tests are all passed.

On phase-4 the last phase, once the SRT tests are passed, the Secure Shield CBT will get the code, and try to break it. Since this is the last security test done by the team, it has to make sure the code is appropriately secured for the required level of intruder, or at least the best it can be. If the Secure Shield CBT determines that the protection is adequate, then the software is ready to be deployed. On the other hand, if the Secure Shield CBT determines that the software is not adequately protected, it will send its results to the Secure Shield Intruder Analysis Team which will analyze these results and send the code back to the appropriate security development team (SSP, SPM, THL TMS, OR SSC), depending on the findings, to be done again. And the cycle in the methodology begins again at that point (i.e. Phase-1 to 4)

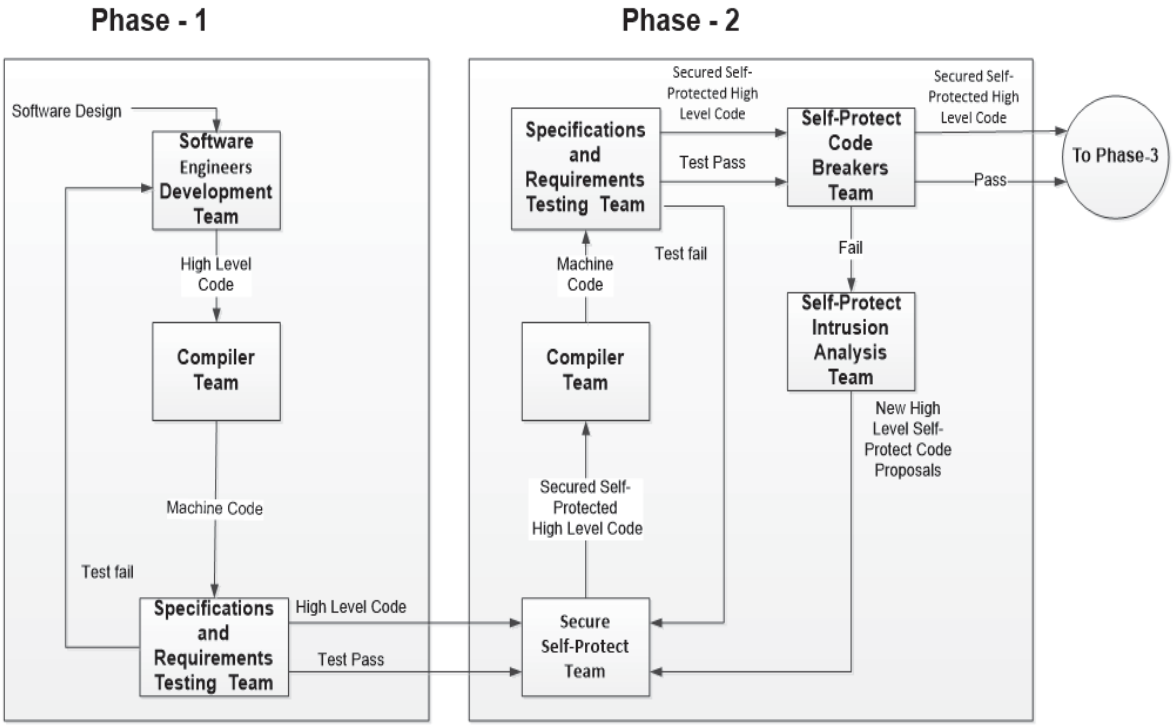

Fig. 1 Phase 1, Phase 2

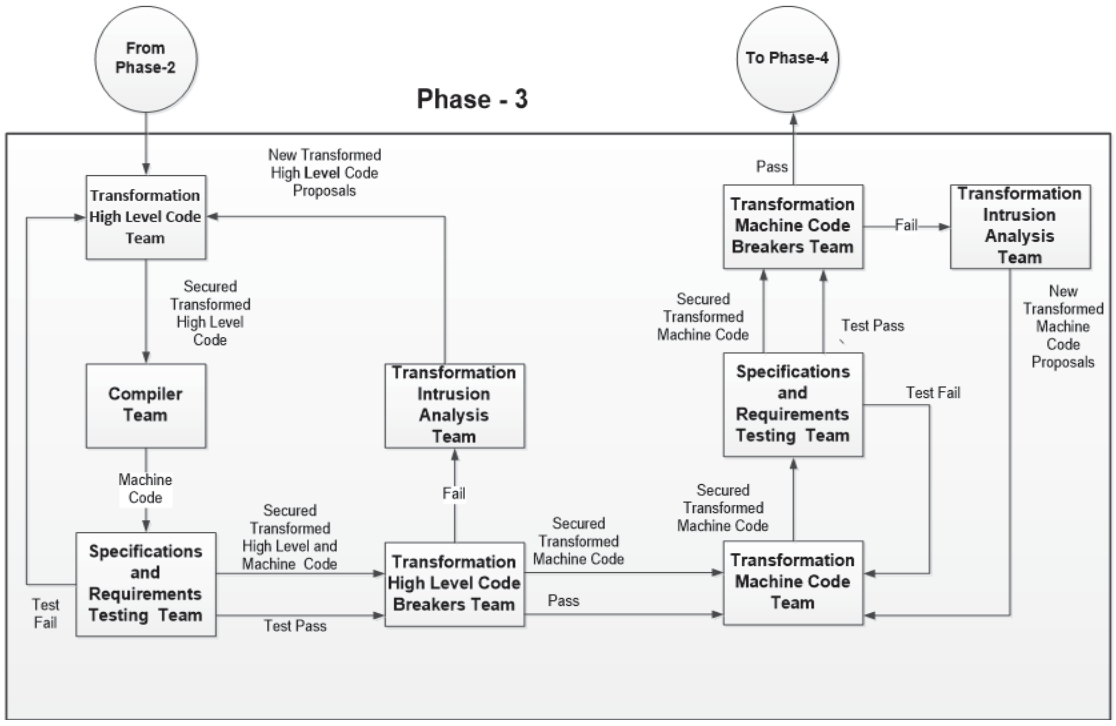

Fig. 2 Phase 3

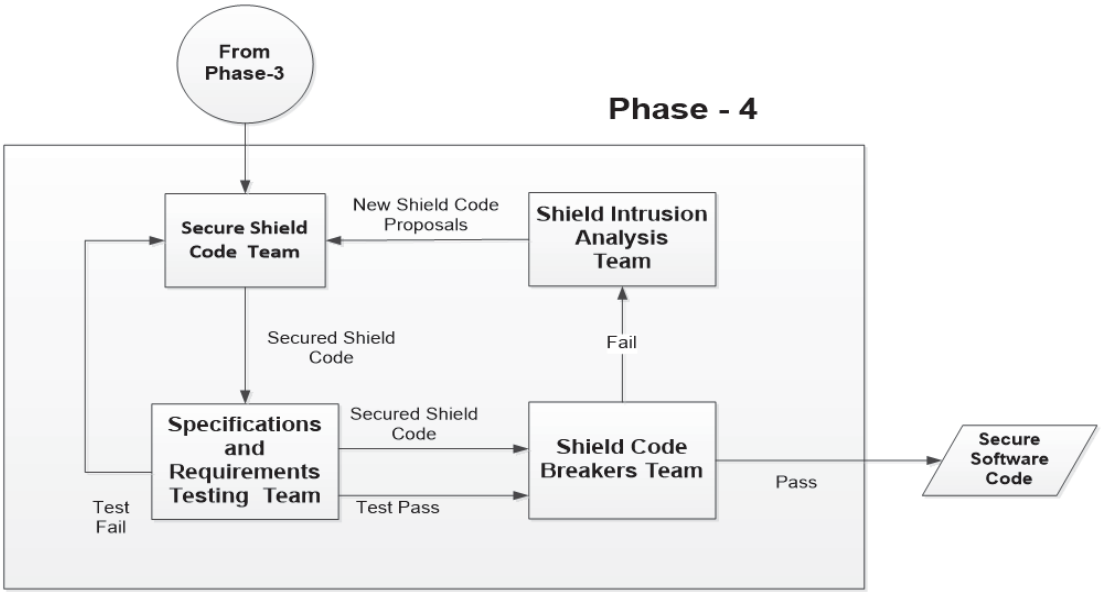

Fig. 3 Phase 4

IV. Final Recommendations

- If this methodology is used for secure software development, we recommend the following general guidelines:
- Isolate the various teams (mainly the SED, SRT, SHL and SMT teams). They should be in different buildings or different cities, which is better.
- Make sure that the team leader for each team is an expert in the field, and knows the whole methodology approach.
- Make sure that the customer, as well as all the members of the SED team, understand the cost of late changes. The later in the methodology that a change occurs, the more expensive it becomes to do it.
- For the Code Breakers Teams, it is recommended to include at least one of each of the SED and one member of each of the secure developing teams. The idea behind this is: we should test that even if an intruder has some knowledge of the contents of the software, the intruder should not be able to break the code.
- For the Intrusion Analysis Team, make sure you select the best personnel you have in secure software. Make sure also, that they are innovators and have a long range vision of the secure software field. This secure software may run for 20, 30 or more years, and we want it protected.
- Protect locally the source code and its documentation.
- Protect locally the algorithms and procedures used in both obfuscations. If an intruder gets to know what algorithms you used for obfuscation, then reverse engineering for the intruder could become much simpler.

V. Conclusions

The main contribution of this paper is the description of a new approach to develop secure software following software engineering concepts.

Our methodology recommends attention to the security of the software at the earliest stage of the project as possible. What we propose that is different from other methodologies is doing the security development at a later phase of the development. By doing these phases we are able to accomplish three major goals in a project: Use sound Software Engineering techniques for phase one of the development; second, minimize the cost of changes and or modifications of the software; and finally, we recommend the use of experts in security to do the development of phase-2 to 4, instead of having developers with some notion of security do it.

The Four Phase methodology provides a more robust and easy to maintain software. The introduction of Code Breakers Team (made out of software security experts) to test the security of the software independently of the software development team in conjunction with the Intrusion Analysis Team is an innovation in methodologies for the development of secure software.

References

[1] Carlos Gonzalez, Ernesto Liñan, "A Software Engineering Methodology for Developing Secure Obfuscated Software", IET Software, Submitted, Sep-2015.
[2] David Chaboya, (20 Jun 2007) State of the Practice of Software Anti-Tamper. Air Force Research Labs Anti-Tamper and Software Protection Initiative (AT-SPI) Technology Office.
[3] Keller, John. "Anti-tamper technologies seek to keep critical military systems data in the right hands – Military & Aerospace Electronics". Militaryaerospace.com. April-26-2010.
[4] Carlos Gonzalez, "User Detection in Secure Self-Protected Software", Unpublished Research, Sep 2015.
[5] Denning, Dorothy E., "An Intrusion Detection Model," Proceedings of the Seventh IEEE Symposium on Security and Privacy, May 1986, pages 119–131.
[6] Scarfone, Karen; Mell, Peter (February 2007). "Guide to Intrusion Detection and Prevention Systems (IDPS)". *Computer Security Resource Center* (National Institute of Standards and Technology) (800–94).
[7] Shields, Tyler (2008-12-02). "Anti-Debugging Series - Part I". Veracode. Retrieved 2009-03-17.

[8] Barak B., O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, "On the (Im)possibility of Obfuscating Programs", pp. 1–18, Advances in Cryptology– Crypto 2001, Springer LNCS 2139 (2001).
[9] Collberg C., C. Thomborson, D. Low, "A Taxonomy of Obfuscating Transformations", Technical Report 148, Dept. Computer Science, University of Auckland (July 1997).
[10] Collberg C., C. Thomborson, D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", Proc. Symp. Principles of Programming Languages (POPL'98), Jan. 1998
[11] Collberg Christian, "Surreptitious Software Exercise, Attacks, Breaking on System Functions", Department of Computer Science, University of Arizona, February 26, 2014.
[12] dreamincode.net, "A Simple Introduction to Obfuscated Code", http://www.dreamincode.net/forums/topic/38102-obfuscated-code-a-simple-introduction/, November 25, 2007.
[13] Cullen Linn, Saumya Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly" http://www.cs.arizona.edu/~debray/Publications/disasm-resist.pdf, Retrieved 2015-06-17.
[14] Feiman Joseph, "Runtime Self Protection: A Must Have, Emerging Security Technology", Gartner Group, 24 April 2012.
[15] Gary McGraw, "Software Security: Building Security In", Addison-Wesley Professional, 2006.
[16] Kenneth R. Van Wyk, Diana L. Burley, Mark G. Graff, Dan S. Peters, "Enterprise Software Security: Design Activities", Addison-Wesley Professional, Dec 31, 2014.
[17] William Stallings, Lawrie Brown, "Computer Security: Principles and Practice", 3rd Edition, Pearson, Jul 8, 2014.
[18] Patterson David, Armando Fox, "Engineering Software as a Service: An Agile Approach Using Cloud Computing", Strawberry Canyon LLC, 2013.
[19] Pressman Roger S., Bruce R Maxim, "Software Engineering: A Practitioner's Approach", 8th edition, McGraw Hill, 2014.
[20] Somerville Ian, "Software Engineering", 9th edition, Addison-Wesley, 2011.
[21] McConnell Steve, "Code Complete: A Practical Handbook of Software Construction", 2nd Edition, Microsoft, 2004.
[22] Fowler Martin, Kent Beck, John Brant, William Opdyke, Don Roberts, "Refactoring: Improving the Design of Existing Code", Boch Jacobson Rumbaugh, 1999.
[23] Aucsmith D., "Tamper Resistant Software: An Implementation", Proc. 1st International Information Hiding Workshop (IHW), Cambridge, U.K. 1996, Springer LNCS 1174, pp. 317-333 (1997).
[24] Kenter Arjan, "Obfuscation" http://www.kenter.demon.nl/obfuscate.html, Retrieved February 22, 2014
[25] Mateas Michael; Nick Montfort. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics". Proceedings of the 6th Digital Arts and Culture Conference, IT University of Copenhagen, 1–3 December 2005. Pp. 144–153.
[26] Toshio Ogiso, Sakabe Yusuke, Soshi Masakazu, Miyaji Atsuko, "Software Obfuscation on a Theoretical Basis and its Implementation", IEEE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, January 2003, 176-186.
[27] Amit Sahai, et al., "Candidate Indistinguishability Obfuscation and Functional Encryption for all circuits", http://eprint.iacr.org/2013/451.pdf, 2013
[28] Amit Sahai and Brent Waters "How to Use Indistinguishability Obfuscation: Deniable Encryption, and More", http://eprint.iacr.org/2013/454.pdf, 2013