On Self-checking Software Components

Ramya Dharam
Department of Computer Science
University of Memphis, TN, USA
rdharam@memphis.edu

Sankardas Roy
Department of Computer Science
University of Memphis, TN, USA
sroy5@memphis.edu

Dr. Sajjan G. Shiva
Department of Computer Science
University of Memphis, TN, USA
sshiva@memphis.edu

ABSTRACT
Software protection has recently attracted interest from developers. Self-checking software is one of the approaches for software protection. This paper surveys the existing techniques for building self-checking software and presents a taxonomy of the current endeavors. This taxonomy should help the reader to get the global view of the current solution space of self-checking software techniques. To the best of our knowledge, our paper is the first attempt to make a survey of this field of study.

Keywords: self-checking software, software protection, hash value, encryption, decryption, checksum.

1. INTRODUCTION
Despite the considerable effort over the last two decades to protect the cyber space, hacking endeavors still grow in numbers and sophistication, which strongly indicates that the system administrators need to take a game-changing strategy. We have to accept the fact that there is no panacea to overcome the ever-growing plethora of cyber security problems.

We envision a holistic security approach which suggests the system administrator to make a thorough analysis of the security threat to the whole system, instead of securing the system part by part. We advocate taking a 4-layer security approach illustrated in Figure 1. The layers are defined as follows: (i) At the innermost layer are the core hardware and software components. We envision each of these components having a provision of being wrapped with a self-checking module (with inspiration from the traditional BIST architecture). We call them Self-checking HW/SW components. (ii) At the 2nd layer lies the traditional network security infrastructure which is built using techniques such as cryptographic algorithms. (iii) At the 3rd layer reside the secure applications which are designed with Built-In or Bolt-On security approaches utilizing self-checking concepts and components. (iv) We envision a game theoretic decision module at the top layer which has the responsibility of choosing the best security strategy for all the inner 3 layers. We observe that in the past the research community has put majority of their effort only for the 2nd and the 3rd layers. We have recently written one paper on designing the top layer, whereas this paper considers design of the innermost layer. In particular, this paper focuses on how to build self-checking software components.

Software is a collection of programs developed to perform the functions as desired by the user. In many cases, software does some of the critical operations for applications like financial transactions, e-commerce etc. The integrity of the programs in these applications must be maintained.

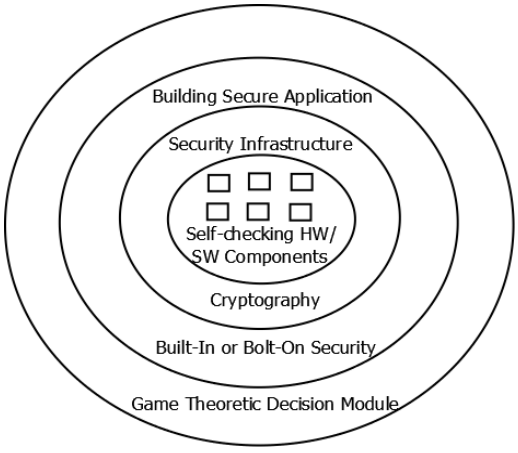

Figure 1: A Holistic Security Approach: The innermost circle represents the hardware and software components. The second innermost circle represents the security infrastructure. The third innermost circle represents building secure application and the outer circle represents game theoretic decision module.

Self-checking software is a software protection approach where the software checks itself to make sure that it has not been tampered so that the software can execute normally. Over the past few years various techniques have been proposed for implementing self-checking software. Self-checking techniques can be static or dynamic. In static self-checking technique, the program checks itself whether it has been modified or not only once before the execution of the program. In dynamic self checking technique, the program checks itself whether it has been modified or not repeatedly during the execution of the program.

In this paper we perform a survey of existing techniques for implementing self-checking software, build a taxonomy of the existing techniques to classify the different existing techniques.

The paper is organized as follows: Section 2 builds taxonomy of the various self-checking software techniques. Section 3 concludes this paper.

2. A taxonomy of existing techniques for self-checking software
We thoroughly surveyed the existing self checking software techniques and found that there are four major techniques to implement self- checking software. According to the survey made we are going to build a taxonomy of different existing techniques in this paper. Figure 1 shows the taxonomy of existing techniques in which each inner node represents the different existing

techniques for self-checking software. Each leaf rectangular node contains the names of the prior research papers that have proposed the corresponding technique.

The four major techniques are Checksums, Hashing, Encryption and Digital Signature. Checksum is a value and can be calculated in different ways using different algorithms. Algorithms like CRC (Cyclic Redundancy Checks) can be used. To verify the integrity of the data, checksums are calculated and compared. If checksums do not mach then we infer that the data might have been modified. Hashing is the technique of converting a larger length value into a shorter fixed length value that represents the original value. Hashing is performed using a hash function which returns a hash value. Data integrity can be maintained by comparing the hash values of data. If they have the same hash value then the data is not altered else the data is altered. Encryption is the technique of converting the data into a form that cannot be easily understood. To convert the encrypted data back to the original form so that it is understood a decryption key is needed. If the data is not decrypted properly then garbage code will be generated so the execution of the program is not possible and the program halts. Digital signature is the technique that uses asymmetric cryptography to encrypt the data. An encryption technique in which public and private keys are different is termed as asymmetric cryptography. Private key is used to create a digital signature of the data. This digital signature will be decrypted with the public key and will be compared with the original data to verify the integrity and the authenticity of the data.

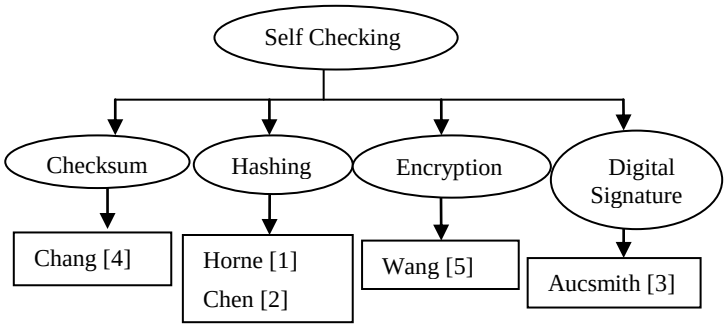

Figure 2: Taxonomy of existing techniques for self-checking software. Each inner node represents the existing techniques for self-checking software. Each rectangular leaf node represents the names of the papers that have proposed the corresponding technique.

The terms such as program, code refers to the original program which is to be protected. These terms are used interchangeably to discuss the techniques proposed in the prior papers.

Horne et al. [1] proposed a dynamic self checking technique. In this technique the executable code is divided into overlapping *intervals (code segments)*. Testers are code segments in assembly language and each tester is randomly assigned to the intervals and is executed as they are encountered. The assembly code in the testers computes the hash value for its assigned interval and compares the result with the correct value which is stored in the component named corrector. Each Interval will have components named *Testers* and *Correctors.* If there is any change in the interval being tested then it will return an incorrect hash value and a response mechanism will be triggered. The overlapping of the intervals and assignment of the testers to many intervals provides a high degree of security. The advantage of this technique is that to avoid the detection of change of even a single bit of code requires disabling all the testers for that particular interval which is a difficult task. The limitation is that placing of testers into the code requires manual effort.

Chang et al. [4] proposed a methodology in which protection of program is obtained by smaller segments of code called guards. Group of guards can work together in which either multiple guards can provide protection together or one guard can protect another guard to form a network of guards. The guards are embedded within the program. Each guard will be programmed to do a specific security related task during program execution. The code in the guard may be either the *checksum code* or *repair code*. The checksum code calculates the checksum of the *guarded code (program)* at runtime and checks if program has been tampered. If found tampered the working of the program will be stopped or other actions could take place. But if no alteration in the program is found the normal execution of the program is continued. The repair code will check the program before its execution and if the program is found to be tampered then the repair code will remove all the changes made to the program and execute the program normally. The strength of this technique is that the user specifies the regions of the program that should be guarded. The shortcoming is that mixing the guard code with the program has to be done with great care in order to avoid the attacker from identifying the guard code and altering them.

Chen et al. [2] have described a software verification method called Oblivious Hashing. This method mainly detects the execution trace of the program by using the hashing instructions and calculates the hash value of the execution trace. The Oblivious hashing instructions will be inserted into the syntax tree which is high level representation of the program and generated during the parsing stage of the compilation process. The syntax tree contains information about dependency of the data in the program. Assignment statements and control flow statements are the programming constructs that determine the execution of the program (execution trace). These are the statements that will be hashed and the oblivious hashing instruction computes hash value. Modification in the execution of program will generate a different hash value so an error is detected and will not allow the further execution of the program. The advantage of this technique is that hash value computation is based on the execution of the program. The limitation is that if some part of the program is not executed then no hash value computation is performed on that part and it remains unprotected.

Aucsmith et al [3] have proposed a tamper resistant technique using Integrity Verification Kernel (IVK). According to the proposed technique, IVK are segments of code that are inserted into the larger program in order to verify the integrity of the larger program. IVK are divided into cells (or code segments) of equal size. IVKs are encrypted and will be decrypted as it executes. Each IVK calculates the digital signature of the program in which it is embedded. The integrity of the program is verified by comparing the calculated digital signature with the correct value that is hard coded into the IVK. IVKs are constructed using specialized tools and they will be inserted into the reserved area of the larger program. One IVK can check other IVK and thus provide the interlocking mechanism. A program can have more than one IVK. The advantage of this technique is that it eliminates

single point failure which ensures that failure of any one IVK will be detected by other because IVKs are interlocked. The shortcoming is that each IVK must contain private key that is used to generate the digital signature.

Wang et al. [5] proposed a mechanism that makes use of multi blocking encryption technique to protect the program integrity in the software. In this mechanism the program is divided into several different sized blocks. The number of blocks could be either the whole program as one block or one instruction per block. The blocks are independent of each other i.e. one block does not have a jump or branch instruction to other block and all the jumps and branch must be within the same block. Each block will be encrypted with different key. The hash value of the blocks is used as the encryption key. The decryption of the next block is done by using the hash value. If the attacker tries to modify the code in the block then it will produce a different hash value and this hash value will not allow to decrypt the next block so the program cannot continue to run and crashes. The point of failure occurs far away from the point of detection which makes it hard for the attacker to find out the exit point. The strength of this technique is that different hash values are used for encrypting different blocks. However one limitation is that for the application of this technique the program must have a tree-like control flow i.e. the program must have only a single entry point.

Other Technique: Castro et al. [6] have presented a technique that prevents the software attacks by enforcing data-flow integrity. During the program execution some data such as return address and function pointers are loaded to program counter and attacks to these data are termed as control-data attacks. Non-control-data attacks are the attacks that do not alter the control flow of the program. Data-flow integrity technique proposed prevents both these attacks. In this technique *static data-flow graph* is computed using *reaching definition analysis*. The task of reaching definition analysis is to map the program instructions to *definition identifier* and will find out the set of all the definitions which reads the value and will construct a graph called static data flow graph. A table called *Run Time Definition Table (RDT)* will be updated by the program with the identifier of the instructions that writes the value. The program also checks if the identifier of the instruction that writes the value which is being read by another instruction is an element of the graph. If the identifier is not the part of a graph means it is an unknown identifier and raises an exception when this data-flow integrity is violated. The tampering of the RDT table by attacker is also prevented. Any attempt to write to the RDT generates an exception. When the program execution is deviated from the data-flow graph computed an exception is raised and will not allow the further execution. The advantage of this technique is that it does not contain any hash value comparisons or encryption techniques. The shortcoming of this technique is the effort needed in maintaining and updating of the RDT table.

Comparing the techniques for Self-Checking software:

The following table compares the different techniques proposed for self-checking software. The dimensions used for comparison are techniques used, vulnerability to run-time attacks and single point failure.

Run-time attacks are the type of attacks in which the adversary is able to successfully find that instruction which performs comparison actions and succeeds in modifying the execution of the software.

Single point failure occurs in the following scenario: if the attacker is able to bypass even one checking instruction in the software code, the whole software will be compromised.

Table1: Comparison of different techniques

Work	Technique Used	Run-time attacks vulnerable	Single point failure
Aucsmith (1996)	Digital Signature	No	No
Chang (2001)	Checksum	No	No
Horne (2002)	Hashing	Yes	No
Chen (2002)	Hashing	Yes	Yes
Wang (2005)	Encryption	No	No
Castro (2006)	Data-Flow Graph	No	No

3. CONCLUSION

This paper provided a survey of the different techniques that were invented over the past few years to implement the self-checking software. The taxonomy presented in this paper will give the reader a global view of the current solution space of self-checking software techniques.

4. REFERENCES

[1] B. Horne, L. Matheson, C. Sheehan, R. Tarjan, *Dynamic Self Checking Techniques for Improved Tamper Resistance*, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer 2320, pp.141-159, 2002.

[2] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M.H. Jakubowski, *Oblivious Hashing: A Stealthy Software Integrity Verification Primitive*, 5th International Workshop on Information Hiding, Noordwijkerhout, The Netherlands, Springer LNCS 2578, pp. 400-414, 7-9 October, 2001.

[3] D. Aucsmith, *Tamper Resistant Software: An Implementation*, Information Hiding, First Int'l Workshop, R.J. Anderson, (ED.), pp. 317-333, May, 1996.

[4] H. Chang and M. J. Atallah, *Protecting Software Code by Guards*, Proc. 1st ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, Pennsylvania, Springer LNCS 2320, pp. 160-175, November, 2001.

[5] P. Wang, S. Kang, and K. Kim, *Tamper Resistant Software Through Dynamic Integrity Checking*, SCIS 2005 The 2005 Symposium on Cryptography and Information Security Maiko Kobe, Japan, Jan.25-28,2005 The Institute of Electronics, Information and Communication Engineers.

[6] M. Castro, M. Costa, T. Harris, *Securing Software by Enforcing data-flow integrity*, Proceedings of the 7th symposium on Operating systems design and implementation, Seattle, Washington, pp. 147-160, 2006.