DFITS: Data Flow Isolation Technology for Security –Security Software Development Environment–

Ryotaro Hayashi Fukutomo Nakanishi Hiroyoshi Haruki
Yurie Fujimatsu Mikio Hashimoto
Corporate Research & Development Center, TOSHIBA Corporation
{ryotaro.hayashi, fuk.nakanishi, hiroyoshi.haruki,
yurie.fujimatsu, mikio.hashimoto}@toshiba.co.jp

ABSTRACT

We propose a security software development environment DFITS for implementing a software which is running on a secure processor architecture. Given a source code, DFITS system analyzes a data flow of the source code focusing on cryptographic operations and automatically classifies every data according to protection requirements. Then, DFITS outputs executable program such that the data to be protected is encrypted by the secure processor, and the data to be accessed (not to be protected) from other entities is located on an unprotected memory. By using DFITS system, programmers can develop security software both securely and efficiently.

Keywords

secure processor, cryptographic protocol, data flow, programming, type inference

1. INTRODUCTION

Due to the remarkable improvement of software and digital contents distribution mechanisms, software have came to be used in various places and there is a growing need for software protection technologies. The purpose of software protection is to prevent (the confidential data of) the software from both reading and falsifying by malicious users, and to guarantee the correct behavior of the software.

A secure processor, such as XOM [1], AEGIS [2], and LMSP [3], offers a possible answer to the problem. It provides a protected memory area for confidential data, on which the confidential data of the software is encrypted and cannot be accessed from other entities. On the other hand, it also provides an unprotected memory area which anyone can access, since considerable amount of data, for example input and output, belonging to the software should be accessed from other entities, while the confidential data should not. In order to develop software running on a security processor, the programmer has to classify every data into the confidential data and the data to be accessed from other entities, and assigns appropriate memory area to every data according to the classification.

However, this is not an easy task. First, the specialist knowledge of security is required. All programmers do not always have such specialist knowledge. Second, there exist enormous amount of the variables (data), including internal and temporary variables, in source code. Therefore, it is difficult and mistakable to assign appropriate memory to every data even for a security specialist. Third, once the unprotected memory is mistakenly assigned to the confidential data, it is hard to detect such potential vulnerabilities. Since such insecure memory assignment does not affect the behavior of the software in normal operation, it cannot be detected by a functional test. Such insecure memory assignment becomes obvious only after the confidential data on the unprotected memory is found and cracked by adversaries.

For solving the problems described above and implementing software both securely and efficiently, we propose a security software development environment DFITS (Data Flow Isolation Technology for Security). DFITS system analyzes source code, classifies every data according to the protection requirement automatically, and outputs executable program with appropriate memory assignment. The name "Data Flow Isolation Technology for Security" is derived from the function to isolate the confidential data by using the security data flow analysis.

By using DFITS system, programmers can develop security software without human errors with respect to the memory assignment. Furthermore, even programmers who are not security experts can implement security software with appropriate memory assignment by using DFITS.

2. DFITS

2.1 Software development flow with DFITS

We consider the situation that a programmer, given a specification of a security protocol, develops software (an executable program) which is running on a security processor. Here, we assume that the (security) specification has been verified by some way, for example a formal security verification (proof).

The programmer has to assign appropriate memory area to every data (variable), and in such a process the programmer faces the problems described above. DFITS is a technology to solve the problems and support appropriate memory assignment.

We consider a development flow of the security software with DFITS as shown in Figure 1.

In the first phase, the programmer generates a source code focusing on the functionality of the software. We assume that the executable program passes the functional test and vulnerabilities such as buffer overflow are removed in this phase. We also assume that the programmer uses the cryptographic function library which consists of the cryptographic functions, such as encryption, signature, hash functions, etc. The functions in this library are designed by

CCS'09, November 9-13, 2009, Chicago, Illinois, USA.

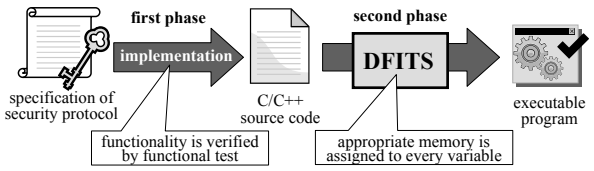

Figure 1: The development flow of the security software with DFITS.

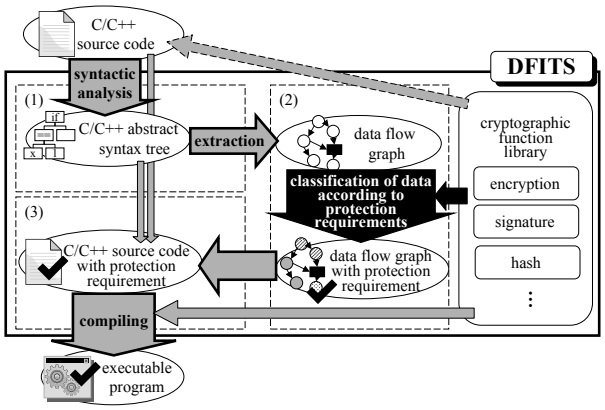

Figure 2: The functional diagram of DFITS system.

security specialists, and implemented with secure memory assignment, that is, appropriate memory is assigned to every local variable of the functions.

In the second phase, the programmer applies DFITS to the source code. DFITS generates an executable program where appropriate memory is assigned to every variable of the source code, including internal and temporary variables.

2.2 The functional diagram of DFITS system

Considering the development flow described above, we design DFITS system as shown in Figure 2. We explain how to analyze the source code and assign memory to data in DFITS system.

First, an abstract syntax tree is generated from an input C/C++ source code by syntactic analysis, and a data flow is extracted by analyzing the abstract syntax tree.

Next, DFITS system classifies data (variables) according to the protection requirements by the security data flow analysis focusing on the cryptographic processing. We briefly explain how to classify the data in Section 2.3.

Finally, DFITS system generates the source code with the additional information for memory assignment according to the classification in the previous phase, and generates the executable program by compiling the derived source code with the cryptographic function library.

2.3 Classification of data according to the protection requirements

In this section, we briefly explain how to classify every data according to the protection requirements.

2.3.1 Protection attributes

The data is classified according to the protection requirements by DFITS method from the viewpoint of whether the data must have the **confidentiality** and the **integrity**.

Table 1: The protection attributes.

memory area	protection attribute	confidentiality	integrity	usage example
unprotected	exposed	×	×	input / output
protected	verified	×	✓	public key
	concealed	✓	×	unverified decrypted-message
	confidential	✓	✓	secret key

The confidentiality ensures that the information is accessible only to those authorized to have access. The integrity is the assurance that the data is consistent and correct. In DFITS method, security data types called *protection attributes* are employed for classification. We show the protection attributes defined in DFITS method in Table 1.

2.3.2 Attribute inference algorithm

DFITS method contains an attribute inference algorithm. This algorithm solves a kind of constraint satisfaction problem where the constraint conditions are as follows.

1. The protection attributes of the inputs and the outputs of the cryptographic functions are restricted.
2. For the other processing, for example substitution, copy, and non-cryptographic processing such as addition and multiplication, all variables within each processing must have the same protection attribute. For example, for the assignment "$a = b + c$", the protection attributes of a, b, and c must be the same.

As mentioned in Section 2.1, in DFITS method the cryptographic function library is provided in which the functions are implemented with appropriate memory assignment. This library consists of the cryptographic functions. For these functions, the protection attributes of the inputs and outputs of the functions (*cryptographic function attributes*) can be defined by analyzing the processing from the viewpoint of the security. For example, since the secret key for decryption must have both the confidentiality and the integrity, it is limited to be stored in the *confidential* variable. The public key for encryption is limited to be stored in the *verified* variable, since the data has to be read by the external entity, and have the integrity.

Since the cryptographic protocol to be implemented consists of a combination of cryptographic functions, the inference algorithm employs the first rule and infers the protection attributes of the data in the whole data flow (source code) based on those of cryptographic functions. Further, by using the second rule, data is isolated according to the protection attributes.

Though the above attribute inference algorithm is based on straightforward requirements for cryptographic and input/output data flow, it occasionally fails in attribute inference from existing cryptographic protocols. Those include PGP, S/MIME, and SSL 3.0 as far as we found. In order to accommodate proposed algorithm to such cases, an additional algorithm is introduced. This is based on the analysis of a relation with two (or more) branch data flow, and the algorithm detects the target variable (e.g. hash input) whose integrity is verified *implicitly* by checking a *related* variable (e.g. hash output) has the integrity. By using this algorithm, the protection attributes for PGP, S/MIME, and SSL 3.0, which have a data flow where such target variable is used as a confidential data, can be determined appropriately. Due to lack of space, we omit details of the algorithm.

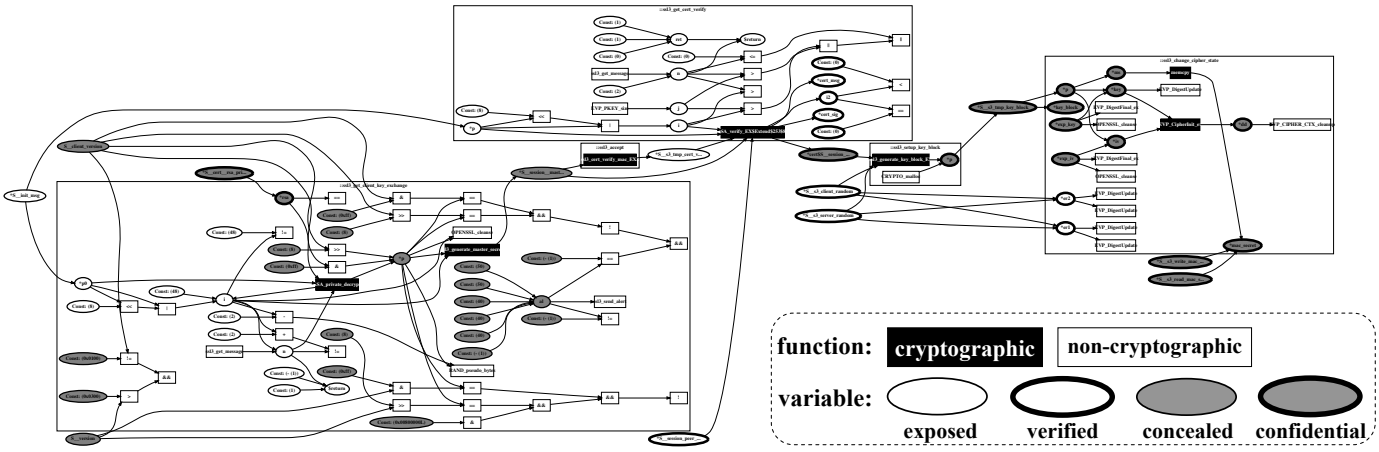

Figure 3: The data flow and the protection attributes of OpenSSL generated by DFITS system.

3. A PROTOTYPE IMPLEMENTATION AND EXPERIMENTAL VALIDATION

We make a prototype implementation of DFITS system for validation of DFITS method. Here, we explain our prototype implementation. (See also Figure 2.)

For generating an abstract syntax tree, we employ the syntax analysis library Elsa [4]. A data flow is extracted by scanning the abstract syntax tree and looking at substitutions and function calls.

In order to classify data according to the protection attributes, in our implementation, constraints are extracted from the data flow, and a typical type inference algorithm is employed as an attribute inference algorithm.

For generating source code with the protection attributes, we convert the abstract syntax tree into source code by means of the pretty printer of Elsa. Furthermore, for generating the executable program with the appropriate memory assignment, the protection attributes are implemented via class templates of C++ with suitable memory allocators. The executable program can be generated by using the C++ complier (g++).

Then, we apply our prototype implementation to a part of source code of OpenSSL, which is an implementation of SSL protocol 3.0 (SSL 3.0). More concretely, parts of `s3_srvr.c` and `s3_enc.c` of OpenSSL 0.9.8g are analyzed. In this experiment, we only focus on whether the attribute inference algorithm succeeds.

Before applying DFITS system, we adapt the target source code to our prototype implementation. First, in the target source code, there exist some variables which are used for several purposes. For example, a variable is used for both a confidential data and an input-output. Since this is not suitable from the security viewpoint and DFITS system cannot deal with such a variable, we remove reuse of variables from the target source code. Second, our existing prototype implementation of DFITS system cannot analyze either function pointers or structure variables. It also cannot deal with interprocedural analysis. Therefore, we adapt the target source code to our prototype implementation so that a resulting source code does not have such structures and has the same functionality as that of the original target source code. Then, the number of lines of the resulting source code is 673.

We defined 7 cryptographic function attributes for the functions within the range of the analysis by DFITS system, and apply DFITS system to the source code. DFITS system outputs (the data flow and) the protection attributes for 108 variables as shown in Figure 3. DFITS system can analyzes not only inputs and outputs of the cryptographic functions, but also internal and temporary variables which are not explicitly appeared in the specification of SSL 3.0. We have manually confirmed that DFITS system appropriately outputs the protection attributes of at least 108 variables, including internal and temporary variables, in the target source code.

4. DISCUSSION AND FURTURE WORKS

One of our future works is to prove the validity of DFITS method formally. Information flow analysis [5] is well-known as a research to guarantee the software security by using the static analysis of source code, similar to DFITS. It is considered that the security analysis for the confidential information in DFITS is simpler (not so rigorous) than that in the information flow analysis. We consider that formal methods for proving the security, such as information flow analysis, will help with the formal validation of DFITS.

It is also our future work to improve DFITS in aspects of both theory and implementation for applying DFITS system to much more cryptographic protocols. For example, the improvement of implementation is required for analyzing function pointers and structure variables in source code.

5. REFERENCES

[1] D. Lie et al, Architectural support for copy and tamper resistant software. *ASPLOS-IX*, pp. 168–177.

[2] G. E. Suh et al, AEGIS: Architecture for tamper-evident and tamper-resistant processing. *ICS 2003*, pp. 160–171.

[3] M. Hashimoto et al, Secure Processor Consistent with both Foreign Software Protection and User Privacy Protection. *SPW2004*, *LNCS* 3957, pp. 276–286.

[4] S. McPeak et al, Elkhound: A Fast, Practical GLR Parser Generator. *CC 2004*, *LNCS* 2985, pp. 73–88.

[5] D. Volpano et al, A Sound Type System for Secure Flow Analysis. *J. of Comp. Sec. 4*, 2-3, pp. 167–187.