Proceedings of the IASTED International Conference
ADVANCES IN COMPUTER SCIENCE AND TECHNOLOGY
January 23-25, 2006, Puerto Vallarta, Mexico

SOFTWARE OBFUSCATION FROM CRACKERS' VIEWPOINT

Hiroki Yamauchi, Yuichiro Kanzaki, Akito Monden,
Masahide Nakamura, Ken-ichi Matsumoto

Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama
Ikoma Nara 630-0192, Japan
{hiroki-y, yuichi-k, akito-m, masa-n, matumoto}@is.naist.jp

ABSTRACT
Various kinds of software obfuscation methods have been proposed to protect security-sensitive information involved in software implementations. This paper proposes a cracker-centric approach to give a guideline for employing existing obfuscation methods to disrupt crackers' actions.

KEY WORDS
Software Protection, Attacks, Tamper-resistance, Symmetric-key Cryptography, Information Leakage

1. Introduction

Software obfuscation has become an essential means to hide secrets involved in today's software systems. Obfuscations transform a program so that it is more complex and difficult to understand, yet is functionally equivalent to the original program [8]. The secrets in a program may include subroutines, algorithms and constant values that are valuable and/or related to system security. For example, decryption algorithms and/or decryption keys involved in digital rights management (DRM) software need to be hided from a *cracker*, a software user who tries to analyze the software to extract its secret information [1][4].

There are various types of software obfuscation methods including control flow obfuscation [6][16][25], inter-module call relation obfuscation [17], identifier obfuscation [24], self-modifying code [10], data obfuscation [7][20], etc. Unfortunately, it is unclear how effective these obfuscations are in protecting confidential information inside a program. It is because "complex" is not equal to "difficult to crack". In addition, many of obfuscations do not assume a clear *threat model*, a model of cracker's behavior to break the security.

This paper seeks for a guideline for employing existing obfuscation methods to protect programs against a skilled cracker who tries to extract security-sensitive data. We focus on the *crackers' viewpoint* including crackers' knowledge, cracking tools and conjectures concerning a target program. We illustrate how to eliminate clues that the cracker may find using the cracking tools. This is a cracker-centric approach focusing on "disrupting" the cracker's actions, while conventional obfuscation methods often fell into a protector-centric approach focusing on building a "complex" program.

In this paper we set our target on a cipher program that involves a symmetric cipher algorithm and a cryptographic key in it. Such a program is commonly used in typical DRM software. We define a security goal and a threat model for the cipher program. Then, based on the model, we illustrate a guideline to employ existing obfuscation methods to protect secret keys included in the program.

2. Security Goal

The security goal for a cipher program is to protect a cryptographic key K contained in a cipher program P without significant performance degradation. Although there exists a powerful protection method called *white-box cryptography* [4][5] to hide the key K, its application area is quite limited since it requires a large memory space (several megabytes) and imposes a serious performance penalty. Thus, using the white-box cryptography is out of scope of this paper.

The major application area, in which this security goal is required, is a development of multimedia player software, since most of player software employs a symmetric cipher scheme, such as DES, AES, C2, etc., to decode enciphered media contents with a cryptographic key (a certain integer value) embedded in the player itself. Such player software is being developed not only for high power PCs but also for low power consumer gadgets including PDAs, mobile phones and mobile game consoles. Therefore, we need to protect the key K by some "light weight" security techniques that do not cause serious performance degradation.

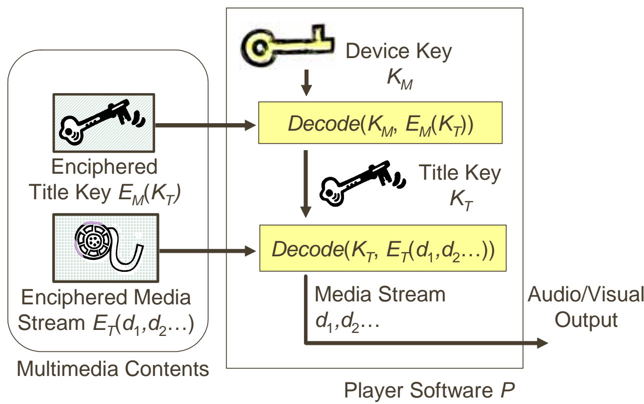

Fig. 1. A mechanism of a typical multimedia player

Figure 1 shows a simplified mechanism of typical multimedia player software. The player software P contains a key K_M called a *device key* (or a *master key*), which is required to be completely hidden from software users (i.e. potential crackers), although K_M is inherently contained in P. This key K_M is used to decode an enciphered title key $E_M(K_T)$, then the decoded title key K_T is used to decode the enciphered media stream $E_T(d_1,d_2, \ldots)$. Usually, the key K_M is supplied to each software company (manufacturer) by some licensing entity with an agreement (permission) to build a player containing the key K_M. However, once K_M had revealed to a user, it becomes possible for the user to build a media ripping program that can produce a raw (non-enciphered) media stream file of an arbitrary (enciphered) media contents. Since such raw media files easily spread out all over the world via pear-to-pear software, there is a strong need to protect K_M from being spied out from P. Indeed, the keys for the CSS encryption standard for DVD media content were revealed by a cracker in 1999 [20]. As a result, programs that overturn DVD copy protection are now widely distributed through the Internet.

3. Threat Model for Cipher Program

When we consider employing a security mechanism to achieve any security goal, we must have a realistic threat model, a model of what a cracker is able (and not able) to do in the real world. For example, a cracker may have an executable (binary) program and an algorithmic understanding of the principles of a cipher used in the program. Also, it will be reasonable to assume that the cracker has a static analyzer such as a disassembler and a decompiler, as well as a dynamic analyzer (debugger) with "breakpoint" functionality.

In [15], Monden et al. characterized a cracker's knowledge and resources along three dimensions, (1) understanding level of a protection mechanism being used, (2) skill level of system observation, and (3) skill level of system control. These dimensions seem useful for evaluating any software protection mechanism; however, in this paper, the dimension (1) is presently out of the scope since we do not assume any particular protection method yet.

Besides the dimension (1), we need to characterize the cracker's understanding level of a cipher algorithm being used because in modern DRM systems, such as CPPM/CPRM, cipher algorithms are open to public while cryptographic keys are kept secret [1].

There are several possible levels for each dimension, e.g. a cracker at low level is an end-user with very limited technical skill and ability, while a high level cracker has a debugger and good technical skills. In this paper, we assume of a top level hacker who has full knowledge of existing cracking tools, computer systems, and cipher algorithms. Below characterizes the supposed cracker along three dimensions (labelled A, B, C).

(A) Algorithm Understanding

The cracker has full knowledge of the principles of a cipher algorithm used in the cipher program P if the specifications of the algorithm were open to public. On the other hand, the cracker does not know the secret information (device keys, s-box, etc.) supplied to P's manufacturer by a licensing entity.

(B) System Observation

The cracker owns a binary file, disassembled code and/or decompiled code of P, as well as a computer system M in which P is executed. The cracker has a debugger with breakpoint functionality that can observe internal states of M, e.g. memory snapshot of M, audio-visual outputs of M and the input and output value of P. The cracker also observes the execution trace of P, i.e. a history of executed opcodes, operands and their values.

(C) System Control

The cracker operates the keyboard and mouse inputs of M, as it executes P with an arbitrary input. The cracker can change the instructions in P as well as the operand values and the memory image of M in any desired way, before and/or during running it on M.

Under our definitions above, crackers have various avenues of attack. They might inspect disassembled code of P and find a portion of code that implements a particular part of a cipher algorithm. This may lead to narrow down the area in P where the secret key is manipulated. They also might observe a stack memory to find candidates of a secret key pushed onto the stack memory during execution [2]. Furthermore, they might collect multiple execution traces of different inputs, and compares them to find candidates of a fixed key appeared in operand values that are insensitive to the inputs [26].

Anyway, since the attack depends on the actual cipher algorithm used in P, we put our target on a C2 Java program and describe more specific and detailed threat model in the next Section.

4. Threat Model for C2 Java Implementation

The C2 (cryptomeria cipher) algorithm is used in CPPM (Content Protection for Prerecorded Media) / CPRM (Content Protection for Recordable Media), which was developed by 4C companies (Intel, IBM, Matsushita and Toshiba) [1] to provide protected exchange of audio-visual content on removable storage media, e.g. DVD-R/RW/RAM, SD Memory Card and Secure CompactFlash. Based on the experience of CSS encryption standard's failure, security aspects of CPPM/CPRM has been enhanced (e.g. key revocation scheme). However, still, a device key must be embedded in content players and it must not be revealed to users. Hence, protecting C2 programs from crackers is still an overarching issue today.

In this paper we focus on the Java implementation (Java class file) of a C2 program (ECB mode) since it is extremely vulnerable to attacks, as it is an intermediate language code, not far from source code representation. If we could defend attacks to the Java implementation, then it would be much easier to defend attacks to other language implementation (such as x86 binary code).

4.1 Algorithm understanding

The outline of the C2 algorithm is shown in Figure 2. C2 is a Feistel network-based block symmetric cipher just like DES. The box "F" indicates the Feistel function. Figure 3 shows details of the Feistel function. The major distinction point is that, C2 uses arithmetic addition and subtraction while DES does not.

Since the specification of C2 is open to public [1], the cracker should have the following knowledge.

- Round keys k_i are either supplied from a key schedule routine or directly written in P as constant values. In the former case, there exists a device key K in P, and K is supplied to the key schedule routine. In the latter case, both K and the key schedule routine may not exist in P. In this case, the cracker's goal is to find all the round keys.
- The number of rounds (iterations) is 10. So, there are 10 round keys k_1, ..., k_{10}.
- The length of each round key is 32-bit.
- The input block size is 64-bit. A block is divided into L (32-bit) and R (32-bit).
- There is a Feistel function F in P. Either L or R is given to F in each round.
- There is a subtraction expression right after F.

Similarly, as for the Feistel function F, the cracker should have the following knowledge.

- Either L or R is added to a round key. This indicates that an add opcode that takes two 32-bit operands in P is a big clue to find round keys.
- The result of addition X (32-bit) is divided into four 8-bit blocks $x_1,\ldots,x_4$. It can be guessed that this division is done by statements "x_1=(X>>24) & 0xff, x_2=(X>>16) & 0xff, x_3=(X>>8) & 0xff, x_4=X & 0xff."
- The lowest 8-bit block x_4 is translated by a S-box table. Here, the S-box table is a set of 256 8-bit values. This indicates there is an array having 256 elements in P. The translation can be done by a reference to the array, e.g. "S-box_array[x_4]" Values of the S-box table are not open to public in CPPM/CPRM case (They are supposed to be concealed just as the device key and round keys).
- Remaining 3 blocks x_1, ..., x_3 are XOR'ed with 0xc9, 0x2b, and 0x65, respectively. This indicates there exist XOR expression and constant values 0xc9, 0x2b and 0x65 in P. Afterwards, these 3 blocks are rotated leftward 2-bit, 5-bit, and 1-bit, respectively.
- Then, four blocks are concatenated back to 32-bit value. This value is then XOR'ed with 9-bit rotated value and 22-bit rotated value. This indicates there exist "shift" opcode and constant values 9 and 22 in P. It can be guessed that the concatenation was done by an expression "x_1<<24|x_2<<16|x_3<<8|x_4."

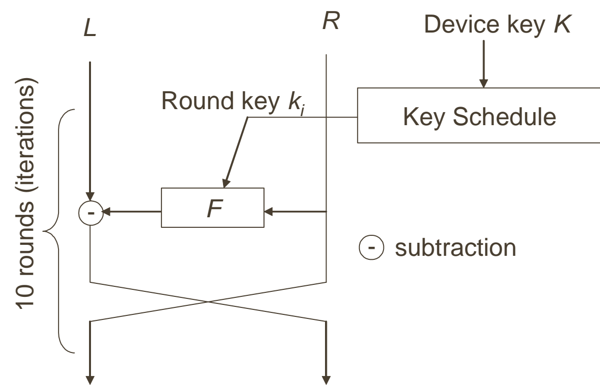

Fig. 2. Feistel Network of C2 cipher

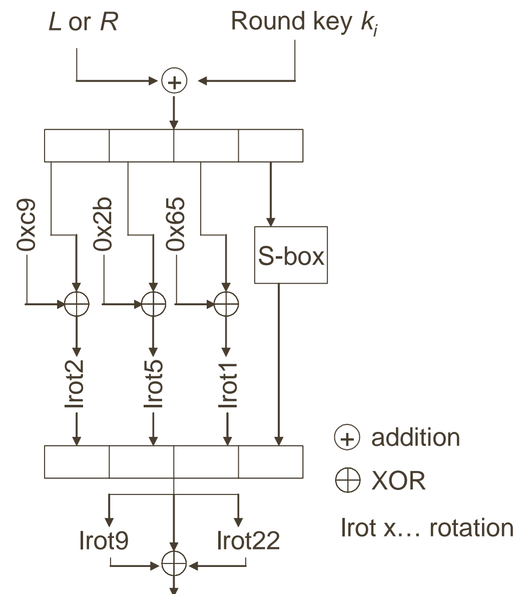

Fig. 3. Feistel Function of C2

Table 1. Output Specifications of tracers injected by AddTracer

Execution	Output
Reference to a local variable	Variable ID or name, its value, line number,
Assignment to a local variable	Variable ID or name, assigned value, line number
Reference to a field	Class name, variable name, its value, line number
Assignment to a field	Class name, variable name, assigned value, line number
In case the variable is an array	Index of an accessed array element
Operation	Opcode, operand type, result of operation, line number
Method invocation	Class name, method name, line number
Return from a method	Class name, method name, line number

```
public static byte lrot8(int x, int n) {
        return (byte)( (x << n) | ( x >>> (8-n) ) );
}
```

Java source code for a rotation of 8-bit value

```
...
x      0x47       assignment    // line 7
n      0x1        assignment    // line 7
x      0x47       reference     // line 7
n      0x1        reference     // line 7
<<     0x8e{int}  operation     // line 7
x      0x47       reference     // line 7
-      0x8{byte}  constant      // line 7
n      0x1        reference     // line 7
-      0x7{int}   operation     // line 7
>>>    0x0{int}   operation     // line 7
|      0x8e{int}  operation     // line 7
...
```

Execution trace of lrot8

Fig. 4. Example of Execution Trace produced using AddTracer

```
rkey[0] = 0x789ac6ee;
rkey[1] = 0x79bc3398;
rkey[2] = 0x48d15d62;
….
```

Source code for defining round keys

```
17:   iconst_0
18:   ldc      #6; //int 2023409390
21:   lastore
22:   aload    8
24:   iconst_1
25:   ldc      #7; //int 2042377112
28:   lastore
29:   aload    8
31:   iconst_2
        ….
```

Disassembled code (by javap –c)

Fig. 5. Disassembled code for round keys

4.2 Tools for system observation and control

The cracker uses disassemblers and decompilers to statically analyze *P*. Sun Microsystems provide java disassembler (known as "javap –c" command). Also, Java disassembler D-Java, which produces jasmin format assembly code [12] is provided by Meyer [13]. Disassembled code can be re-assembled back to class files by jasmine assemblers [14][21]. Various java decompilers are also available although in most cases, they produce imperfect java source code [23].
The cracker also uses debuggers, such as jdb, provided by Sun Microsystems. Also, there is a powerful dynamic tool called AddTracer [22], which injects tracers (monitoring code) all over the Java class file. By executing a Java class file *P* in which tracers has been injected, whole execution trace of *P* can be obtained by simply executing it. Table 1 shows what the tracers output during execution. Figure 4 shows an example of an execution trace produced using AddTracer. Since references and assignments to variables are output with their value, secret keys are revealed if *P* is a naive implementation.

5. Guideline for Applying Obfuscation

In order to achieve the security goal, we need to hide the following information in *P*.

- Round keys $k_1, \ldots k_{10}$ (and a device key K, if it exists)
- S-box table
- Feistel function
- Distinctive opcodes and operands
- Obfuscation itself

Below describes crackers' actions to find above information, and as a guideline of obfuscation, we describe how we can disrupt the actions.

5.1 Obfuscation of round keys

As shown in Figure 5, candidates of round keys can be easily found in disassembled code since the keys are large integers of 32-bit length. Thus, we need to divide the keys into smaller length values. For example, in case a naive implementation of a round key is as follows.

rkey[0] = 0x789ac6ee;

Then, one way to hide the key is to divide them into four 8-bit integers (sub keys) as follows.

```
static byte SecretConstant[] = { (byte)0xB6, (byte)0xAA, ….
```

Source code for defining S-box

```
static {};
 Code:
  0:     sipush   256
  3:     newarray byte
  5:     dup
  6:     iconst_0
  7:     bipush   -74
  9:     bastore
  10:    dup
  11:    iconst_1
  12:    bipush   -86
  14:    bastore
         …
```

Declaration of an array (length=256 type=byte)

1st element

2nd element

Disassembled code (by javap –c)

Fig. 6. Disassembled code for S-box

```
b_rkey[0] = 0x78;
b_rkey[1] = 0x9a;
b_rkey[2] = 0xc6;
b_rkey[3] = 0xee;
```

In case we need to compute with the key, e.g. "rkey[0]^= n", we could compute with the sub keys as follows.

b_rkey[0] ^= n_1;
b_rkey[1] ^= n_2;
b_rkey[2] ^= n_3;
b_rkey[3] ^= n_4;

Where $n = n_1 << 24 \mid n_2 << 16 \mid n_3 << 8 \mid n_4$

In case we need to compute with the original keys, e.g. "t = rkey[n] + *data*", we could compute with the new keys e.g. by "t' = new_rkey[n] + *data* * 4". The original value t can be computed by "$t = (t' - 3) / 4$" when it is needed. Instead of using linear encoding, we could also consider using residue encoding, bit exploded encoding [7], secret sharing homomorphism [3][19], variable merging [9] and the use of error collecting code [11].
Note that an array itself is a big clue to the cracker. Since there are 10 round keys, the cracker might expect that there is an array of 10 elements which hold round keys. We will discuss this issue in the next subsection.

5.2 Obfuscation of S-box

Different from DES and AES, C2 cipher's the S-box table is not open to public in CPPM/CPRM standards. Thus, a manufacturer must conceal it just as round keys.
Since the S-box table of C2 cipher is a set of 256 8-bit values, a naive S-box implementation would be an array having 256 elements (Figure 6). In such an implementation, S-box can be easily found in the disassembled code.

Therefore, we need to use more complex data structure to implement the S-box. One of the simplest ways is to split, fold and/or interleave arrays [9]. Also, a data structure suitable for hiding secret data was recently proposed [18].

5.3 Obfuscation of Feistel function

By using a dynamic analysis tool such as a profiler or AddTracer, the cracker can easily find the Feistel function since it is one of the most frequently executed place in P. If the cracker could locate the place of the Feistel function, it would be quite easy for the cracker to find round keys because they are the inputs of the Feistel function (Figure 3).
One way to reduce the execution frequency of the Feistel function is to prepare a number of different implementations of the Feistel function, and randomly call one of them when needed.
We also recommend not writing the Feistel function as a "method" or a "function." Also, we should not write a loop that calls the Feistel function 10 times, which indicates 10 rounds of a Feistel network. Employing both local- and external-control flow obfuscations [16][25] with adding some randomness to disrupt dynamic analysis would be useful to hide the Feistel network structure.

5.4 Obfuscation of distinctive opcodes and operands

There are several distinctive (conspicuous) opcodes and operands in a naive C2 implementation. For example, XOR opcode, add and sub opcode, bit shift opcode, numerical operands 0xc9, 0x2b, 0x65, 9, 22, 0xff, etc. All these opcodes and operands should be replaced with combination of other opcodes and operands. For example, XOR can be implemented with AND, OR and NOT. Note that hiding conspicuous numerical operands (such as 0xc9) not to appear in P is insufficient. We need to hide them not to appear in an execution trace (e.g. Figure 4) as well.

5.5 Hiding obfuscation

Since obfuscation methods themselves have distinctive features, we need to hide them so that the cracker can not recognize which obfuscation method is being used. For example, if we employ residue encoding [7] to hide round keys, a lot of modulo opcodes will be introduced in P. In this case, we need to write our own (obfuscated) modulo routines instead of simply using module opcodes.

6. Conclusion

In this paper we defined a generic goal and a generic threat model for cipher programs containing secret information. Then, as a case study, we described more specific threat model for C2 cipher programs. Based on

the model, a guideline for applying obfuscation methods was shown.

Although we listed typical avenues of attack and defences in this paper, there is no assurance that we could list all the attacks exhaustively since the crackers might conduct unthinkable avenues of attack. We need to keep on updating the guideline to improve resilience to the attacks. In the future, we also need to implement a cipher program based on the proposed guideline, and conduct an experimental evaluation using expert hackers to evaluate the resilience against attacks.

Acknowledgements

This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 16700033, and Grant-in-Aid for 21st century COE Research (NAIST-IS --- Ubiquitous Networked Media Computing).

References

[1] 4C Entity, Content protection for recordable media specification – Introduction and common cryptographic elements, rev. 1.0, 31 pp., Jan. 2003.

[2] K. Akai, M. Misawa, and T. Matsumoto, Evaluating tamper resistance by searching runtime data, *IPSJ Journal*, Vol.43, No.8, pp.2447-2457, Aug. 2002. (in Japanese).

[3] J. C. Benaloh, Secret sharing homomorphisms: keeping shares of a secret, *Proc. Advanced in Cryptology*, pp.251-260, 1986.

[4] S. Chow, P. Eisen, H. Johnson, P. van Oorschot, A white-box DES implementation for DRM applications, *Proc. 2nd ACM Workshop on Digital Rights Management* (*DRM2002*), *Lecture Notes in Computer Science*, Vol. 2696, pp. 1-15, 2003.

[5] S. Chow, P. Eisen, H. Johnson and P.C. van Oorschot, White-box cryptography and an AES implementation, *Proc. 9th International Workshop on Selected Areas in Cryptography* (*SAC2002*), *Lecture Notes in Computer Science*, Vol. 2595, pp. 250-270, 2003.

[6] S. Chow, H. Johnson, and Y. Gu, Tamper resistant control-flow encoding, *United States Patent* 6,779,114, Filed 19 Aug. 1999, Issued 17 Aug. 2004.

[7] S. Chow, H. Johnson, and Y. Gu, Tamper resistant software encoding, *United States Patent* 6,594,761, Filed 9 June 1999, Issued 15 July 2003.

[8] C. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation – Tools for software protection, *IEEE Trans. on Software Engineering*, Vol. 28, No. 8, pp. 735-746, 2002.

[9] C. Collberg, C. Thomborson, and D. Low, Obfuscation techniques for enhancing software security, *United Sates Patent* 6,668,325, Assignee: InterTrust Inc., Filed 9 June 1998, Issued 23 Dec. 2003.

[10] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, Exploiting self-modification mechanism for program protection, *Proc. 27th IEEE Computer Software and Applications Conference*, pp. 170–179, Nov. 2003.

[11] S. Loureiro and R. Molva, Function hiding based on error correcting codes, *Proc. International Workshop on Cryptographic Techniques and Electronic Commerce* (*CRYPTEC99*), pp. 92-98, July 1999.

[12] J. Meyer and T. Downing, *Java Virtual Machine*, (O'Reilly & Associates, Inc., 1997).

[13] J. Meyer, "D-Java," http://mrl.nyu.edu/~meyer/jvm/djava/

[14] J. Meyer, "Jasmin Home Page," http://jasmin.sourceforge.net/

[15] A. Monden, A. Monsifrot, and C. Thomborson, Tamper-resistant software system based on a finite state machine, *IEICE Trans. on Fundamentals*, Vol.E88-A, No.1, pp.112-122, Jan. 2005.

[16] A. Monden, Y. Takada, and K. Torii, Methods for scrambling programs containing loops, *Trans. of IEICE*, Vol.J80-D-I, No.7, pp.644-652, July 1997. (in Japanese).

[17] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, Software obfuscation on a theoretical basis and its implementation, *IEICE Trans. Fundamentals*, Vol.E86-A, No.1, pp.176-186, 2003.

[18] H.Oguro, T.Iino, Y.Hirai, S.Hakomori, Implementation of multiple precision arithmetic improving resistance against white-box attacks, *The 2005 Symposium on Cryptography and Information Security* (*SCIS2005*), Jan. 2005 (in Japanese).

[19] J. Patarin and L. Goubin, Secret key cryptographic process for protecting a computer system against attacks by physical analysis, *United States Patent* 6,658,569, Filed 17 June 1999, Issued 2 Dec. 2003.

[20] A. Patrizio, Why the DVD hack was a cinch, Wired News, Nov. 1999, http://www.wired.com/news/technology/0,1282,32263,00.html

[21] Soot: A Java optimization framework, http://www.sable.mcgill.ca/soot/

[22] H. Tamada, AddTracer, Injecting tracers into Java class files for dynamic analysis, http://se.aist-nara.ac.jp/addtracer/

[23] The decompilation Wiki of Program-Transformation.Org, http://www.program-transformation.org/Transform/DeCompilation

[24] P. M. Tyma, Method for renaming identifiers of a computer program, *United States Patent* 6,102,966, Assignee: PreEmptive Solutions, Inc., Aug. 2000.

[25] C. Wang, J. Hill, J. Knight, and J. Davidson, Protection of software-based survivability mechanisms, *Proc. International Conference of Dependable Systems and Networks*, pp. 193-202, July 2001.

[26] H. Yamauchi, The evaluation for tamper-resistance of programs based on the instruction sequence differential attack, *Master's Thesis*, *Graduate School of Information Science*, *Nara Institute of Science and Technology*, NAIST-IS-MT0351135, Feb. 2005 (in Japanese).