SCIS 2007 The 2007 Symposium on Cryptography and Information Security
Sasebo, Japan, Jan. 23-26, 2007
The Institute of Electronics, Information and Communication Engineers

C 言語におけるライブラリ呼び出し隠蔽のための名前難読化手法

A Name Obfuscation Method to Hide Library Function Calls in the C Language

玉田 春昭 * 中村 匡秀 * 門田 暁人 * 松本 健一 *
Haruaki Tamada Masahide Nakamura Akito Monden Ken-ichi Matsumoto

あらまし C 言語において，システムコールの名前を隠すための難読化手法を提案する．従来から提案されている関数名を隠すための難読化は名前を定義している部分を変更するため，システムコールや標準ライブラリに含まれる関数など，ユーザが定義できない名前を隠すことができない．これらの関数の定義を変更すると移植性が著しく低下するためである．そこで本稿では，C 言語においてシステムコールや標準ライブラリの関数の名前を隠すことのできる名前難読化手法を提案する．具体的には名前を予め暗号化しておき，実行時に復号してから関数を呼び出すために，動的リンクを用いる．提案手法を用いることで，コンパイルしてできた実行バイナリ中から `printf` などのシステムが定義した関数の名前を隠すことができ，バイナリ中からシンボル名を検索する攻撃からソフトウェアを保護することが可能となる．

キーワード 難読化, ソフトウェア保護, 動的リンク

1 はじめに

ソフトウェアをクラックするための第一段階として，プログラムの実行バイナリ中に含まれるシンボル名を検索する方法がある [5]．例えば，何らかの認証をスキップするためにはまず，authenticate のような認証に関係する単語を実行バイナリ中から探し出す．その後，その周辺を詳細に解析するという方法が考えられる．このような攻撃の第一段階目を防止するための方法の一つに，名前難読化がある [8]．名前難読化はシンボル名に込められた意味を隠すことを目的にした難読化手法の一つである．

従来の名前難読化はプログラム中の名前を別の名前に静的に置換するものであるため，ユーザが定義した関数や変数の名前を変えることは可能であった．しかし，OS が提供するシステムコールや標準ライブラリに含まれる関数の名前を静的に置換することは現実的に不可能である．多くの環境で共通に使われる名前を変更することは，移植性を著しく下げるためである．そのため，クラックの第一段階としてライブラリ関数名の検索が行われた場合，従来の名前難読化手法ではソフトウェアを保護することができなかった．

一方 Java 言語に代表されるオブジェクト指向言語ではリフレクションというプログラム自身の情報を動的に取得・実行できる強力な機構がある．そのため，予め名前を暗号化しておき，実行時に復号して，文字列から動的に名前解決を行うことで，標準ライブラリの名前を隠蔽することが可能である [10]．しかし，C 言語にはリフレクション機構がないために，この手法をそのまま適用することはできない．

そこで本稿では，C 言語においてシステムコールや標準ライブラリ関数の名前を隠すための難読化手法を提案する．提案手法は予め関数名を暗号化しておき，実行時に動的リンクを行い文字列から動的に関数の名前を解決し，実行する．そのため，実行バイナリ中にも難読化対象の関数名が現れることはなく，クラックの手掛かりとなる情報を隠蔽することが可能となる．

2 準備

2.1 ソフトウェア難読化

ソフトウェア難読化とは，あるプログラムを非常に理解しづらい等価なプログラムに変換することで，プログラムを不正なクラックから保護するものである．難読化の概念をより厳密に定義するために，まず，プログラム理解について考える．人間 (攻撃者) がプログラムを理解する際には，様々な認知活動が行われるため，プログラム理解の一般的な定義を与えることは難しい．また，プログラムの何を理解するかで，認知活動は異なる．例え

* 奈良先端科学技術大学院大学 情報科学研究科, 〒 630-0192 奈良県生駒市高山町 8916-5, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama Ikoma Nara, 630-0192 Japan

ば，あるモジュールの機能を理解する場合と，データ構造を理解する場合，特定の関数の場所を理解する場合とでは，理解のプロセスが異なる．このことに注目し，本稿では，プログラムの理解とは「与えられたプログラムから理解の対象となる情報を取り出すこと」と定義する．

定義 1 (プログラム理解) pを与えられたプログラム，Xをpに含まれる任意の情報 (の集合) とする．この時，ユーザがpからXを何らかの方法で外部に抽出 (リバース・エンジニアリング) できた時,「ユーザはXに関してpを理解した」と定義する．この時，ユーザが理解にかかるコストを$cost(p, X)$と書くことにする．ここでコストとは，解析にかかる時間，労力，必要な知識，設備などを含む．

定義 2 (難読化) pを与えられたプログラム，Xを与えられたpの情報 (の集合) とする．また，pの入出力写像を$\mathcal{IO}_p : I \to O$と書く．ここで，Iは全ての入力の集合，Oは全ての出力の集合である．この時,「pのXに関する**難読化**」とは，あるプログラム変換Tをpに適用して，以下の条件を満たすプログラム$p'(= T(p))$を得ることである．

条件 1 $\mathcal{IO}_p = \mathcal{IO}_{p'}$

条件 2 $cost(p, X) < cost(p', X)$

条件 1 は難読化前と難読化後のプログラムが，同一の入出力写像を持つ，すなわち，プログラムの外部的な振る舞いが変わらないことを保証するものである．条件 2 は，難読化後のプログラムp'から情報Xを取り出すことが困難になることを意味する．

2.2 名前難読化

名前難読化は，プログラム中に現れる名前 (識別子) を別のものに付け替える難読化である．プログラム言語における名前は，計算機にとっては単なる識別子であるため，名前の付け替えがプログラムの挙動に影響を与えることはない．しかし，名前は人間にとってプログラムを理解するための重要な手がかりとなる [3]．従って，元プログラムの名前を非常にわかりづらい名前に変換することで，難読化を実現する．

定義 3 (名前難読化) pを与えられたプログラム，U_pをpに現れる全ての名前の集合，$N_p(\subset U_p)$を難読化すべき任意の名前の集合とする．pの**名前難読化**とは，pにおける各名前$n \in N_p$を別の名前$n'(= T(n)$とする) に変換し，別のプログラムp'を得る難読化である．ここで，Tは一対一写像$T : N_p \to N_{p'}$ $(N_{p'} \subset U_{p'})$である．

```
 1: #include <stdio.h>
 2: void cat(FILE *fp){
 3:   int character;
 4:   while((character = fgetc(fp)) != EOF)
 5:     putchar(character);
 6: }
 7: void cat_file(char *filename){
 8:   FILE *fp = fopen(filename, "r");
 9:   if(fp != NULL){
10:     cat(fp);
11:     fclose(fp);
12:   }
13:   else perror(filename);
14: }
15: int main(int argc, char *argv[]){
16:   int i;
17:   if(argc == 1) cat(stdin);
18:   else
19:     for(i = 1; i < argc; i++)
20:       cat_file(argv[i]);
21:   return 0;
22: }
```

図 1: 簡易版 cat コマンド (ファイルを表示する)

プログラムp中に表われる名前とは，主に関数名，変数名のことを指す．さらに，同じ名前でも**定義部** (宣言部) と**使用部** (呼び出し部) の双方に現れる．名前難読化では，pが含む全ての名前から，難読化すべき名前の集合N_pをいかに選択するか，また，名前の変換方法Tをいかに実装するかが鍵となる．

2.3 従来手法

従来の名前難読化手法は，プログラム中の定義部に現れる名前を静的に別の名前に置き換えるものである．具体的には，以下の手順で行われる．

従来の名前難読化の手順

入力: プログラムp，名前集合N_p.

出力: 難読化されたプログラムp'.

手順: 各$n \in N_p$について，以下の操作を行う．結果として得られたプログラムをp'とする．

Step 1: 各$n \in N_p$について，pにおけるnの定義部を別の名前n'に置き換える．

Step 2: pにおけるnの使用部を，Step 1 で置き換えたn'に置き換える．

入力におけるN_pとして，ユーザ (開発者) がpで定義した任意の**ユーザ定義**の名前を与えることが出来る．図 1 の C 言語による簡易 cat プログラムを従来の名前難読化手法を用いて難読化した例を図 2 に示す．

この例では，関数名をそれぞれ `cat` → `a`，`cat_file` → `d` に，ローカル変数名を `fp` → `b`，`character`→`c`，`filename` → `e`，`argc` → `f`，`argv` → `g` に変更している．

上で述べた手法は，比較的実装が簡単であり，難読化後のプログラムの性能劣化が少ないため，広く実用化さ

```
 1: #include <stdio.h>
 2: void a(FILE *b){
 3:   int c;
 4:   while((c = fgetc(b)) != EOF)
 5:     putchar(c);
 6: }
 7: void d(char *e){
 8:   FILE *b = fopen(e, "r");
 9:   if(b != NULL){
10:     a(b);
11:     fclose(b);
12:   }
13:   else perror(e);
14: }
15: int main(int f, char *g[]){
16:   int i;
17:   if(f == 1) a(stdin);
18:   else
19:     for(i = 1; i < f; i++)
20:       d(g[i]);
21:   return 0;
22: }
```

図 2: 従来の名前難読化

れている．例えば，Java 言語用の Dash-O[6] や.Net 向けの Dotfuscator[7] があり，この他にも数多くの名前難読化ツールが存在する．

しかしながら，この手法は**システム定義**の名前に適用できないという大きな制限がある．システム定義の名前は汎用ライブラリや標準ライブラリなどで定義される名前であり，p においては使用部のみに現れる．システム定義の名前は，様々な環境で汎用的に利用される．従って，p においてこれらを難読化してしまうと，p のポータビリティを著しく下げてしまうため，実用的ではない．図 2 の例では，`fgetc` や `putchar`，`fopen`，`fclose`, `perror` といった名前を別名に変更することはできない．これらの関数名は `stdio.h` で定義されており，変更すると別の環境でコンパイルが不可能になるためである．

このように，従来手法では，システム定義の名前を隠蔽できないため，攻撃者にプログラム理解の手掛かりを残してしまうことになる．また，静的に置換された名前は再置換も容易なため，難読化を解除されてしまうおそれがある．従って従来の名前難読手法は，難読化としてはあまり強力ではない．

3 提案手法

従来手法の問題点を解決すべく，システム定義の名前を難読化可能な新たな手法を提案する．

3.1 キーアイデア

2.3 で述べたとおり，基本的にシステム定義の名前は変更すべきではないため，プログラム中から別名で呼び出す (使用する) ことはできない．もちろん，ソースコード上で違う名前にしておき，プリプロセッサで元の名前に置換することもできるが，プリプロセス後，元の名前に戻ってしまうため，コンパイル後の実行バイナリには元の名前が含まれてしまう．また，システム定義の関数のラッパを作成し，そのラッパ関数を呼び出すことで，呼び出し元からは元の関数名を隠すことはできるが，やはり当該関数を呼び出している事実はバイナリ中から隠すことはできない．よって，従来の名前難読化手法によってソースコード中のシステム定義の名前を静的に置換することはできなかった．これに対し，本研究では**動的名前解決**という機構を導入する．プログラム中で使用部に表われる名前を予め別の名前に変換 (暗号化) しておき，プログラム実行時にその名前参照がある度に元の名前に戻す機構である．

C 言語では関数の呼び出しに現れる名前は，コンパイル時に静的に決定されている必要がある．しかし，ブラウザなどの大規模なアプリケーションでは必ずしも全ての機能が静的に解決されているわけではない．プラグインというライブラリの形で，後から機能を追加することが可能であることが多い．そのために，C 言語には**動的リンク**という機構が備わっている．プラグインは実行時に動的にリンクされ，その機能が呼び出されるのである．本研究ではこの動的リンクの機構を動的名前解決を実現する手段として用いる．ただし，動的リンクを行うことができるのは共有ライブラリに対してのみであるため，提案手法で難読化できる名前は共有ライブラリに含まれた名前に限定される．

3.2 動的名前解決

動的名前解決は，プログラム中の使用部に現れる名前を**文字列として**予め暗号化しておき，実行時，必要に応じて元の名前に戻すアプローチである．いま，難読化すべき名前の集合を N_p とする．ただし，N_p はプログラム p において，使用部に現れる関数名であり，かつ，パス d が表す共有ライブラリで定義されているものとする．また，各 $n \in N_p$ と共有ライブラリのパス d に予め暗号 E を適用し，得られた名前を $n'(= E(n)), d'(= E(d))$ とする．そして，実行時には以下の手順で動的名前解決を行う．

動的名前解決の手順

Step 1: 文字列 d' を復号し，共有ライブラリのパス d を得る．

Step 2: d を動的にロードし，共有ライブラリのハンドル h を得る．

Step 3: 文字列 n' を復号し，元の関数名 n を得る．

Step 4: h から文字列 n を使って関数 n のポインタ s を得る．

Step 5: s を使って関数 n を実行する．

3.3 動的名前解決を用いた名前難読化

提案する名前難読化の手順は以下の通りである．ただし，以下において N_p は使用部に現れるシステム定義の関数名とする．

提案する名前難読化の手順

入力: プログラム p，名前集合 N_p.

出力: 難読化されたプログラム p'.

手順: p に以下の手順を適用する．得られたプログラムを p' とする．

Step 1: 任意の文字列暗号 E を用いて各名前 $n \in N_p$ を暗号化する．得られた集合を $N'_p = \{n' | n' = E(n)\}$ とする．

Step 2: 各 $n \in N_p$ について，p における n の使用部を 3.2 で述べた手順を行う処理に変換する．

4 実装

4.1 動的リンク

ここでは，提案手法を実現するための実装について述べる．提案手法実現のために共有ライブラリを実行時にロードし，実行したい関数を動的にリンクする必要がある．そのために，dlopen，dlsym，dlclose を使うことができる [11].

dlopen は引数に指定された共有ライブラリをロードし，そのハンドルを返す．引数の文字列に「/」が含まれていた場合，絶対パス，もしくは相対パスを指定したことになる．また，渡す文字列に「/」が含まれていなければ環境変数 LD_LIBRARY_PATH などで指定されたディレクトリから検索することもできる．

dlopen で得られた共有ライブラリのハンドルから共有ライブラリに含まれるシンボルの参照を得る関数が dlsym である．dlsym を使い関数シンボルの値を取得して関数ポインタに代入することで，関数を動的にリンクすることができる．

dlopen を用いて動的リンクを使って関数を実行する例を図 3 に示す．例では libm.so に含まれている余弦を求める cos 関数を動的リンクしている．まず 8 行目で dlopen 関数を使い libm.so をロードしている．第 2 引数の RTLD_LAZY は必要に応じてシンボルを解決するように処理を遅らせるための値である．次に 13 行目で dlsym 関数を使い，8 行目で得た共有ライブラリのハンドルと"cos"という文字列から cos 関数のポインタを得る．18 行目で関数ポインタ経由で cos 関数を呼び出し，最後に 19 行目でロードした共有ライブラリを閉じている．

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <dlfcn.h>
 4: int main(int argc, char *argv[]){
 5:   void *handle;
 6:   char *error;
 7:   double (*cosine)(double);
 8:   handle = dlopen("libm.so", RTLD_LAZY);
 9:   if(handle == NULL){
10:     printf("%s: %s\n", "libm.so", dlerror());
11:     return 1;
12:   }
13:   cosine = dlsym(handle, "cos");
14:   if((error = dlerror()) != NULL){
15:     printf("dlsym error %s: %s\n", "cos", error);
16:     return 1;
17:   }
18:   printf("%f\n", cosine(atof(argv[1])));
19:   dlclose(handle);
20:   return 0;
21: }
```

図 3: dlopen, dlsym, dlclose の使用例

4.2 復号鍵と復号アルゴリズム

提案手法で難読化を行ったとき，復号アルゴリズムと復号鍵が実行バイナリに含まれてしまう．暗号化された文字列を復号するためには復号アルゴリズムと復号鍵が必須であるためである．そのため，クラッカーの元に実行バイナリしかない場合であっても，暗号化された文字列を復号することは不可能ではない．

これを解決するため，提案手法で難読化後，自己書き換えを用いて復号鍵や復号アルゴリズムをカムフラージュ（偽装）する方法が考えられる [12]．具体的には (A) 予め復号鍵を本来のものとは異なる鍵にしておき，実行時に自己書き換えを用いて本来の鍵に戻す，(B) 予め復号ルーチンを変更しておき，実行時に自己書き換えを用いて本来の復号ルーチンに戻す，の二つの方法が有効である．また，自己書き換え以外にも，アンチデバッガ技術 [2] を併用することが望ましい．

4.3 ユーザ定義の関数名の隠蔽

これまで述べてきた難読化手法は共有ライブラリの関数のみに適用できる手法であり，ユーザが定義した関数に対してはそのまま適用することはできない．ユーザ定義の関数に提案手法を適用するにはユーザ定義の関数を共有ライブラリにする必要がある．共有ライブラリでなければ動的にリンクすることができないためである．

ただし，最近の gcc, glibc, binutils を用いると，ユーザ定義の関数を共有ライブラリにすることなく動的リンクすることができる．これらは PIE (Place Independent Executable) をサポートしており，gcc のコンパイルオプションに -fPIC -pie を渡すことで任意のアドレスにロードが可能な実行ファイルを作成することが可能である．この実行ファイルはデータのアクセスやジャンプが相対アドレスとなっており，共有ライブラリと非常に似た性質を持っている．実際に上記のコンパイルオプショ

```
 1: #include <stdio.h>
 2: #include <dlfcn.h>
 3: char *decrypt(char *d, char *s, int k){
 4:   char *c = d;
 5:   for(; *s != '\0'; d++, s++) *d = *s + k;
 6:   *d = '\0';
 7:   return c;
 8: }
 9: void cat(FILE *fp){
10:   int c;
11:   void *h;
12:   int (*a)(FILE *);
13:   void (*b)(int);
14:   char dest[256];
15:   h = dlopen(decrypt(dest, "daZ[&kg&.", 8),
16:              RTLD_LAZY);
17:   a = dlsym(h, decrypt(dest, "^_]l[", 8));
18:   b = dlsym(h, decrypt(dest, "hml[‘Yj", 8));
19:   while((c = a(fp)) != EOF) b(c);
20:   dlclose(h);
21: }
22: void cat_file(char *filename){
23:   FILE *fp;
24:   FILE *(*a)(char *, char *);
25:   void (*b)();
26:   void (*c)(char *);
27:   void *h;
28:   char dest[256];
29:   h = dlopen(decrypt(dest, "a^WX#hd#+", 11),
30:              RTLD_LAZY);
31:   a = dlsym(h, decrypt(dest, "[deZc", 11));
32:   b = dlsym(h, decrypt(dest, "[XadhZ", 11));
33:   c = dlsym(h, decrypt(dest, "eZggdg", 11));
34:   fp = a(filename, "r");
35:   if(fp == NULL) c(filename);
36:   else{
37:     cat(fp);  b(fp);
38:   }
39:   dlclose(h);
40: }
41: int main(int argc, char *argv[]){
42:   int i;
43:   if(argc == 1) cat(stdin);
44:   else
45:     for(i = 1; i < argc; i++)
46:       cat_file(argv[i]);
47:   return 0;
48: }
```

図 4: 提案手法で難読化した図 1 の cat プログラム

ンに加え，動的リンクのためのシンボルを残すオプション `-rdynamic` を gcc に渡すと実行も動的リンクも可能なバイナリを作成することができる [11].

5 ケーススタディ

5.1 難読化例

図 1 のプログラムを提案手法を用いて難読化した例を図 4 に示す．ただし，紙面スペースの都合上エラー処理は省略しており，暗号アルゴリズムは簡単化のためシーザー暗号を用いている．

図 4 では，オリジナルのソースコード (図 1) で使用された全てのライブラリ関数 (`fgetc`, `putchar`, `fopen`, `fclose`, `perror`) の名前が難読化され，現れないことがわかる．また，このソースコードをコンパイルしてできた実行バイナリにも上に述べた 5 つのライブラリ関数の

```
int tak(int x, int y, int z){
  if(x <= y) return y;
  return tak(tak(x - 1, y, z), tak(y - 1, z, x),
             tak(z - 1, x, y));
}
```

図 5: 竹内関数

```
int tak(int x, int y, int z){
  void *handle;
  void (*t)(int, int, int);
  char dest[256];
  int return_value;
  handle = dlopen(decrypt(dest, "ubl", -1),
                  RTLD_LAZY);
  t = dlsym(handle, decrypt(dest, "wdn", -3));
  if(x <= y) return y;
  return_value = t(t(x - 1, y, z), t(y - 1, z, x),
                   t(z - 1, x, y));
  dlclose(handle);
  return return_value;
}
```

図 6: 図 5 のプログラムの難読化例

表 1: `tak` 関数の呼び出し回数

`tak` 関数に渡す引数	`tak` 関数の呼出回数
`tak( 2, 1, 0)`	9
`tak( 4, 2, 0)`	53
`tak( 6, 3, 0)`	673
`tak( 8, 4, 0)`	12,605
`tak(10, 5, 0)`	343,073
`tak(12, 6, 0)`	12,604,861

名前は出現しないことが確認できた．

5.2 難読化のオーバーヘッド

提案手法のオーバーヘッドを測定するため，図 5 に示す竹内関数 (たらいまわし関数)[4] の呼び出しを難読化し，実行時間を計測した．実行環境は Dell PowerEdge 2850, Xeon マルチプロセッサ 3.6GHz, RAM 4GB, linux kernel 2.4.21-47.ELsmp，コンパイルは gcc 3.2.3 20030502 (Red Hat Linux 3.2.3-56) で行い，最適化オプションは O2 とした．

ただし，`tak` 関数はユーザ定義の関数であるため，4.3 で述べたように動的リンク可能な実行バイナリを作成した．難読化したソースコードを図 6 に示す．`tak` 関数に渡す引数ごとそれぞれの呼び出し回数を表 1 に示す．また，実行時間をそれぞれ 10 回計測した平均を図 7 に示す．グラフでは横軸に試行した `tak` 関数の呼び出し，縦軸は実行に要した時間を対数で表している．

`tak` 関数が 1 万回呼ばれるのに要した時間はオリジナルが 0.137 ミリ秒，難読化したプログラムが 10.48 ミリ秒であった．この結果から動的リンクを行うと約 100 倍遅くなることがわかる．

6 関連研究

名前難読化は自動化が容易であるため，現在流通している難読化ツールのほぼ全てが名前難読化を行っている．

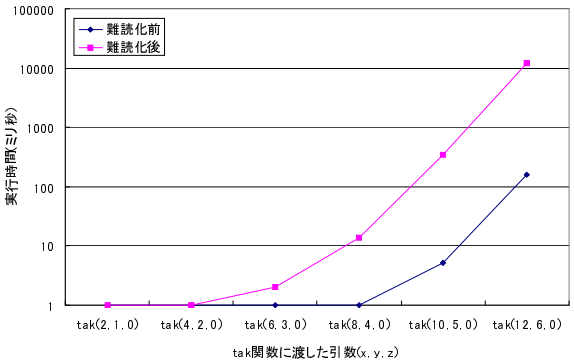

図 7: 竹内関数の難読化前後の実行時間

C/C++用の難読化ツールとして，cobf がある [1]．cobf は予約語や標準ライブラリ関数の名前を別の名前に置き換え，プリプロセッサで元の名前に戻している．しかし，ソースコード上での名前は難読化されているものの，バイナリ中から隠すことができたわけではない．他のツールにおいても，基本的に名前の定義部を書き換えるため，標準ライブラリ関数を使っている事実を隠蔽する目的に使えなかった．提案手法では，関数の使用部で関数名を暗号化し，動的にリンクを行うことで，この問題を解決している．なお，従来手法と提案手法は併用することができる．その場合はまず従来法によってユーザ定義の名前を難読化した後，提案手法で難読化することになる．

また，刑部らはオブジェクト指向の多態性（ポリモーフィズム）を用いて，異なるメソッドを同一の名前にする難読化手法を提案している [9]．この手法も名前難読化の一つとして数えられるが，定義部の名前を変更しているだけであるため，やはりシステムが提供する名前を変更することができない．

7 まとめ

本稿では C 言語において従来隠すことのできなかった標準ライブラリなどの関数の名前を隠すための難読化手法を提案した．プログラムの実行時に暗号化された名前を動的リンクを用いて解決することにより従来法では隠すことのできない標準ライブラリなどのユーザが定義できない関数を隠すことができる．

今後の課題として，より実践的な攻撃に対する耐性の調査，また，提案手法の枠組みの中で復号アルゴリズムや復号鍵を隠す手法について研究を行う．

参考文献

[1] Bernhard Baier. COBF — the C/C++ Sourcecode Obfuscator, January 2006. http://home.arcor.de/bernhard.baier/cobf/, accessed 10 December 2006.

[2] Silvio Cesare. Linux anti-debugging techniques (fooling the debugger), January 1999. http://www.uebi.net/silvio/linux-anti-debugging.txt, accessed 29 December 2003.

[3] B. D. Chaudhary and H. V. Sahasrabuddhe. Meaningfulness as a factor of program complexity. In *Proc. of the ACM 1980 annual conference*, pp. 457–466, 1980.

[4] Donald E. Knuth. Textbook examples of recursion. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pp. 207–229, 1991.

[5] Kracker's and BEAMZ. クラッカー・プログラム大全—禁断のシリアルナンバー解析テクニック. データハウス, December 2003.

[6] PreEmptive Solutions. DashO —the premier Java obfuscator and efficiency enhancing tool. http://www.agtech.co.jp/products/preemptive/dasho/index.html.

[7] PreEmptive Solutions. Dotfuscator —.Net obfuscator, code protector, and pruner. http://www.preemptive.com/products/dotfuscator/index.html.

[8] Paul M. Tyma. Method for renaming identifiers of a computer program. United States Patent 6,102,966, August 2000. Filed: Mar.20, 1998.

[9] Yusuke Sakabe, Masakazu Soshi and Atsuko Miyaji. Java obfuscation —approaches to construct tamper resistant object-oriented programs, IPSJ Journal, Special Issue on Research on Computer Security Characterized in the Context of Social Responsibilities, **46**, 8, pp. 2107–2119 (2005).

[10] 玉田 春昭, 門田 暁人, 中村 匡秀, 松本 健一. プログラム変換装置，呼出し支援装置，それらの方法およびそれらのコンピュータ・プログラム. 特願 2005-171372, June 2005.

[11] 高林 哲, 鵜飼 文敏, 佐藤 祐介, 浜地 慎一郎, 首藤 一幸. Binary Hacks ハッカー秘伝のテクニック 100 選. オライリー・ジャパン, November 2006.

[12] 神崎 雄一郎, 門田 暁人, 中村 匡秀, 松本 健一. 命令のカムフラージュによるソフトウェア保護方法. 電子情報通信学会論文誌, Vol. J87-A, No. 6, pp. 755–767, June 2004.