Some New Approaches For Preventing Software Tampering

Bin Fu*
Computer Science Dept.
University of New Orleans
New Orleans, LA 70148, and
Research Institute for Children
200 Henry Clay Avenue
New Orleans, LA 70118
fu@cs.uno.edu

Golden Richard III
Computer Science Dept.
University of New Orleans
New Orleans, LA 70148
golden@cs.uno.edu

Yixin Chen
Computer Science Dept.
University of New Orleans
New Orleans, LA 70148, and
Research Institute for Children
200 Henry Clay Avenue
New Orleans, LA 70118
yixin@cs.uno.edu

ABSTRACT

In this paper, we propose several methods to increase the difficulty of reverse engineering applications, with special emphasis on preventing the circumvention of copy protection mechanisms that permit only authorized users to execute the applications. We apply the hashing function to transform some constants in the software and recover them during the execution with the correct input of the password. The security of such a method depends on the hardness of the invertibility of the hashing function.

1. INTRODUCTION

Each year software piracy results in billions of dollars in lost revenue. Software designers, in particular those serving vertical markets, need methods for protecting their software from unauthorized use and reverse engineering.

Reverse engineering serves a number of purposes, from defeating software copy protection schemes to program understanding to assisting competitors in duplicating the functionality of a software product. Many vertical market applications (e.g., those in the digital forensics field) where cost of development is high, yet the number of licenses sold is expected to be rather small, are particularly attractive candidates for protection from reverse engineering. A number of general tools are available to reverse engineers. Those include debuggers, which allow dynamic analysis of a program during execution, disassemblers, which generate assembler code from executables, and decompilers, which attempt to recreate high-level source code which roughly corresponds to the original source code of an application. In addition, a number of specialized tools are available, which allow monitoring systems calls executed by an application, file activity, and modification of system data areas such as the Windows registry. Preventing reverse engineering of applications executing on current computing platforms is essentially impossible. Therefore the goal of any "anti" reverse engineering technique is to substantially increase the amount of work that a reverse engineering attempt entails, hopefully beyond the useful lifetime of a software application (or a particular version of the application).

A number of techniques have been proposed to make reverse engineering more difficult. These include: code obfuscation (e.g., [2, 5, 10, 11, 13, 16, 17, 19]), physical protection of the computing platform (e.g., Trusted Computing Platform Alliance (TCPA) [18]), encrypting executables (e.g., Armadillo system, which is changed to SoftwarePassport now), watermarking (e.g., [6, 7]), and the use of secure coprocessors (e.g., [15, 1, 4, 9, 12, 14]). Many techniques for protection of software against reverse engineering involve source code transformations, such as the introduction of aliases into transformed source code, making static analysis very difficult, since alias detection is known to be NP-Hard. Machine code obfuscation is also useful to prevent reverse engineering. Such obfuscation techniques include changing executable layouts, splitting data areas, changing the layout of arrays, and unrolling loops. The problem with such techniques is that many sophisticated tools are available to the reverse engineering community, and in situations where only a small block of code is of interest (e.g., code which accesses a USB dongle to verify a software license or forces an user to enter a password), the techniques may be ineffective.

Though software obfuscation is perhaps the most active research area addressing software protection, attackers can still easily detect code where authorization-to-use checks (e.g., the entry of passwords or checking for an attached hardware device) are located and bypass these checks. Therefore some additional work needs to be done. In [3], some negative results were presented about the existence of very general obfuscator, which is expected to obfuscate every program. Nevertheless, we can still develop efficient techniques to obfuscate many concrete computer programs.

One common method for preventing unauthorized software access is to provide a serial number, password, or hardware dongle to each legal user, and check it whenever the software is installed or executed. This method is very easy to break. On the other hand, the password approach is still attractive since it is familiar to users and easy to use. This paper presents two novel methods for protecting soft-

*Bin Fu is supported by the Louisiana Board of Regents fund under contract number LEQSF(2004-07)-RD-A-35.

ACM SE'06 March 10-12, 2006, Melbourne,Florida,USA

ware via passwords. Both methods involve modifying critical program constants. Our methods are easy to implement and difficult to attack if an adversary does not have detailed information about the algorithms and corresponding implementations used in a software product. In sections 2-5, we discuss the basic methods.

In section 6, we apply our methods to the software obfuscation. We show how constants can affect the flow of the program. Applying our method for changing and recovering constants, we make the program flow confusing.

2. METHOD 1: OBFUSCATING MULTIPLE CONSTANTS

The first method is based on a very simple observation that some constants can significantly affect the operation of an application. If those constants are not correctly set, the software will not be able to run correctly. A constant c is selected, and is changed to c-hash(password), where hash is a hashing function. When the software starts, it expects to read the password from the user. Then c-hash(password) is changed back to c by adding hash(password) to c-hash(password) after the password is read. If the the user inputs an incorrect password, c will not be recovered correctly.

2.1 Our method by example

Let's start with a simple example that explains our method. Our software is for solving quadratic equation $x^2+bx+c=0$, which has two roots $\frac{-b+\sqrt{b^2-4c}}{2}$ and $\frac{-b-\sqrt{b^2-4c}}{2}$. We may design the following C-source code to solve the equation.

```
#include <stdio.h>
#include <math.h>
#define realPassword 2314
#define c1 2.0
#define c2 4.0

void quadratic(double b, double c, double *root1, double
*root2){
double temp;
temp=sqrt(b*b-c2*c);
*root1=(-b+temp)/c1;
*root2=(-b-temp)/c1;
}

void main(){
double b,c, root1, root2;
int password;
scanf("%d", &password);
if (password! = realPassword)
printf("password is incorrect");
}
else {
scanf("%lf, %lf",&b,&c);
quadratic(b,c, &root1, &root2);
printf("%lf, %lf",root1, root2);
}
}
```

When an attacker gets this program, he can easily identify the area checking the password, and remove it. Then he gets the code for solving the quadratic equation. The source code also releases the password information. If function hash(x) is a hashing function that generates an integer number, we can convert the source code into the following form. Assume that hash.h permits us to call function hash(x). Notice that d1 and d2 are assigned values e1=2.0-hash(realPassword)) and e2=4.0- hash(realPassword+1), respectively. The advantage of our method is that the password is merged with the algorithm of the software. It may affect some important constants. If the attacker wants to use the software without the password, he has to understand the algorithm and its implementation to a considerable level. If k constants are selected, and the range of hashing function has m different values, the number of possibilities is m^k. This makes combinatorially impossible to try all of the cases to get the correct constants when $k \geq 3$ or $m \geq 2^{20}$ since the number of cases will be more than $2^{20} \times 2^{20} \times 2^{20} = 2^{60}$. Selecting 100 constants from a software does not affect much its execution speed, but it makes the attacher hard to break.

```
#include <stdio.h>
#include <math.h>
#include "hash.h"
#define d1 e1
#define d2 e2
double c1,c2;

void quadratic(double b, double c, double *root1, double
*root2){
double temp;
temp=sqrt(b*b-c2*c);
*root1=(-b+temp)/c1;
*root2=(-b-temp)/c1;
}
void main(){
double b,c, root1, root2;
int password;
scanf("%d", &password);
c1=d1+hash(password);
c2=d2+hash(password+1);
scanf("%lf",&b);
scanf("%lf",&c);
quadratic(b,c, &root1, &root2);
printf("%lf, %lf",root1, root2);
}
```

A hashing function is expected to have one-way property. It is easy to compute to obtain $h(x)$ for the input x, but it is hard to compute x from the hashing value $y = h(x)$. When dealing with multiple constants, if the attacker knows some of the constants, one of his choices is to invert the hashing function to derive the password. The one way property makes it practically infeasible to invert.

Let $h(x)$ be the hashing function. We select crucial constants $c_1, \cdots, c_n$ from the program. Let p be the password. Compute $d_1 = c_1 - h(p+1), \cdots, d_n = c_n - h(p+n)$.

In the beginning of the program, let q be the password read from the user of the software. The program tries to recover the constants from executing the following codes:

$$\begin{aligned} c_1 &= d_1 + h(q+1); \\ &\cdots \\ c_n &= d_n + h(q+n); \end{aligned}$$

If $q = p$, then those $c_1, \cdots, c_n$ will be correctly recovered.

2.2 Implementation Considerations

An important problem for implementation is the rounding and overflow for processing the constants. In the beginning, it collects some constants from the software, and makes sure that it does not have overflow and rounding problem. We should make sure the transformation will keep the equivalence between the new and old versions. We suggest that it collects those signed and unsigned integers that are less than half of its maximal value.

2.2.1 Pick up the crucial constants

From our experience, a software, especially the large software, usually has some constants. Some of them are closely related to its core algorithm. If the designer of the software sets up the protection for its software, he/she can easily pick up some constants which are important to its algorithm. We suggest to select those constants to which the software is sensitive.

2.2.2 Use some common constants

In the C program, there are a lot of statement like a++. We can transform it into something like a=a+c, where c is initially assigned to 1-hash(realPassword) and is recovered later by increasing it by hash(password) if realPassword=password.

2.2.3 Create new constants

For an integer or float variable x, $x + c_0$ and $x * c_1$ are identical to x if $c_0 = 0$ and $c_1 = 1$. Therefore, we can create some constant c_0 and c_1 and apply them to some variables. The values c_0 and c_1 must be recovered during the execution of the software. Otherwise, the program will fail.

3. METHOD 2: HASHING CONSTANTS IN THE MEMORY

Our second method is to save some constants at a list of memory addresses that can be calculated from a hashing function with the password. The real password is $p = 2357$ with hashing values $hash(p+1) = 4, hash(p+2) = 1, hash(p+3) = 23$, and $hash(p+4) = 62$. The constants $2.0, 4.0, 1.0, 0.0$ should be assigned to c_1, c_2, c_3, c_4 respectively so that the program can run correctly. When the program starts, it puts a series of numbers into the array constants[ ]. The constants 2.0, 4.0, 1.0 and 0.0 are stored at constants[4], constants[1], constants[23], and constants[62], respectively. If the password read from the user is equal to p, then the program will make that $c_1 = 2.0, c_2 = 4.0, c_3 = 1.0$, and $c_4 = 0.0$.

```
#include <stdio.h>
#include <math.h>
#include "hash.h"
#define array_size 64
double c1,c2,c3,c4;

void quadratic(double b, double c, double *root1, double
*root2){
double temp;
temp=sqrt(b*b-c2*c);
*root1=(-b+c3*temp)/c1+c4;
*root2=(-b-c3*temp)/c1;
}

int main(){
double b,c,root1, root2;
int password;
double constants[array_size]={
3.12, 4.0, 5.12, 4.13, 2.0, 5.16, 2.17, 3.0,
7.52, 6.9, 8.73, 9.23, 9.0, 8.42, 7.29, 5.9,
1.92, 9.2, 3.92, 6.63, 8.7, 8.36, 9.15, 1.0,
4.91, 4.9, 7.19, 2.76, 5.8, 8.79, 5.32, 4.9,
9.30, 2.9, 8.17, 9.26, 7.2, 3.12, 3.56, 3.7,
7.98, 6.8, 3.32, 5.78, 4.6, 1.26, 4.32, 2.8,
3.10, 5.3, 3.83, 4.28, 7.9, 3.64, 4.57, 4.9,
2.23, 3.8, 3.87, 6.12, 4.5, 4.98, 0.00, 9.0};
scanf("%d", &password);
c1=constants[hash(password+1)];
c2=constants[hash(password+2)];
c3=constants[hash(password+3)];
c4=constants[hash(password+4)];
scanf("%lf",&b);
scanf("%lf",&c);
quadratic(b,c, &root1, &root2);
printf("%lf, %lf", root1, root2);
return 0;
}
```

To protect a general software, we select a password p and a chunk of memory m[s]. Let $c_1, \cdots, c_n$ be the n crucial constants in the program. Let $h(x)$ be a hashing function. The function $h_1(x) = h(x)\%s$ is a function whose range is between 0 to $s-1$. Therefore, $h_1(x)$ gives valid address in the array $m[\ ]$. The beginning of the program has the statements like

$$\begin{aligned} m[0] &= d_0 \\ m[1] &= d_1 \\ &\cdots \\ m[s-1] &= d_{s-1} \end{aligned}$$

The selected constants $c_1, \cdots, c_n$ are among $d_0, d_1, \cdots, d_{s-1}$. Let's say $c_1 = d_{i_1}, \cdots, c_n = d_{i_n}$. Assume that $i_1 = h_1(p); i_2 = h_1(p+1); \cdots; i_n = h_1(p+n-1)$. Let q be the password that is read from the user. The program tries to obtain those constants from

$$\begin{aligned} c[1] &= m[h_1(q+1)] \\ c[2] &= m[h_1(q+2)] \\ &\cdots \\ c[n] &= m[h_1(q+n)] \end{aligned}$$

If q is the same as the real password p, then $c[1], \cdots, c[n]$ are correctly recovered. Otherwise, they will obtain a new set of values. Such a method uses hashing function to build up a permutation. If k is the number of elements in an array for saving those constants, the number ways for putting n constants is $k \cdot (k-1) \cdots (k-n+1)$.

We need to consider the collision of the hashing function in the range 0 to $k-1$. After p is selected, make some adjustments to avoid those collisions. Assume that

$$\begin{aligned} h_1[p+i_1] &= h_1[p+j_1] \\ h_1[p+i_2] &= h_1[p+j_2] \\ &\cdots \\ h_1[p+j_t] &= h_1[p+j_t] \end{aligned}$$

We let $h_2[p+j_i]$ be $h_1[p+j_i] + b_i$ for some shift $b_i (i =$

$1, \cdots, t$). The method does not change the values of those constants. It only changes the permutation of them.

4. RECOVER FROM THE WRONG PASSWORD

The methods will crash the software if the user does not give the password. A legal user may input wrong password. We should let such a user have the chance to input the password again. We suggest to use the following method. Let $hp = h(p)$. After q is read from the user, we check if $hp = h(q)$. If yes, we start recovering those constants. Otherwise, we ask the user to type the password again a few times. The attacker is unlikely to figure out the correct password p from its hashing value hp as the hashing function is difficult to invert.

5. COMBINE THEM TOGETHER

The method 1 has the substitution consideration, which is widely used in the design of many classical encryption algorithms. The method 2 has the permutation consideration. We can combine the two methods in one implementation. We give an example in the Appendix 1.

5.1 Off line procession

- Select two hashing functions: One hashing function $hash\_address()$ is used for computing the address. The other one $hash\_value()$ is used for adjusting the values of some constants.
- Select constants: Select some constants $c_1, \cdots, c_n$ from the program. From our experience $n = 10$ should be big enough.
- Avoid address collision via hashing: Select some shift values $s_1, \cdots, s_n$ for the address computation via hashing function. It satisfies that $hash\_address(p + s_j) \neq hash\_address(p + s_k)$ for different j and k from $\{1, 2, \cdots, n\}$.
- Select a password: An integer p is selected as the password. We will use the inputs $p, p+s_1, p+s_2, \cdots, p+s_n$ to the hashing functions in the implementation.
- Adjust those constants: The hashing function $hash\_value()$ is used to adjust those constants. Let $c_i' = c_i - hash\_value(p + i)$ for $(i = 1, \cdots, n)$.
- Compute the hashing value of the password $hp = hash\_value(p)$. It is used to hide the password when checking if the user gives the password and giving the user multiple times to type it.

5.2 On line procession

- Put the adjusted constants into an array. It can be implemented in C with syntax like $constants[\ ] = \{d_0, \cdots, d_{s-1}\}$, where $\{c_1', c_2', \cdots, c_n'\}$ is the subset of $\{d_0, \cdots, d_{s-1}\}$. Each c_i' is equal to $d_{hash\_address(p+s_i)}$, which is at the address $constant[hash\_address(p+s_i)]$.
- Read and check the password. After the user provides the password q (it may not be the same as p), it checks if $hp = hash\_value(q)$. If not, the user may get several chances to type the password. If yes, then it continue the execution.
- Recover the constants: Each constant c_i is recovered from $constants[hash\_address(q+s_i)] + hash\_value(q+i)$.

6. APPLICATION TO SOFTWARE OBFUSCATION

In some situations, a reverse engineer may get the source code and analyze the algorithm of the implementation. In order to prevent such an attack from a reverse engineer, we apply our constants transformation methods to the software obfuscation. We show how the constants affect the flow of the program in C. We use an array of function pointers in C programming language. The example for solving the quadratic equation is implemented with function pointers in the Appendix 2. The four constants c_0, c_1, c_2, and c_3 are used to control the function calling flow. The result after obfuscation transformation is shown at Appendix 3. The transformation to two constants c_4 and c_5 are also used to affect the results of arithmetic operations.

For a general software, we need to make an array of function pointers. We would like to make it hard for a reverse engineer to predict the flow of the function calling chain. Different functions may have different parameters list types. In order to put many functions into a function pointer array, we need set up two pieces of memory space. One is for the parameters list, the other is for saving the return values of those functions. For example, assume two functions have the types int f1(int a) and double f2(double b, int c), then we may define global variables int a, double b, and int c to pass the parameters in calling f1 and f2. The functions f1 and f2 will select the parameter values from a,b and c. On the other hand, we also define the global variables int r1, and double r2 for holding the results from f1 and f2. We can define new functions void f1a( ) and void f2a( ) to replace the old f1 and f2, respectively. The result of f1a is put into r1 and the result of f2a is put into r2.

7. AN UNCOMPUTABLE RESULT

In this section we prove it is undecidable to find out the constants to make the program run correctly. Our result is based on computability theory. We will follow the presentation in [8]. This result shows there is no computational way to assign right constants so that the program can run correctly if the program is not changed.

Let $N = \{0, 1, 2, \cdots, \}$ be the set of all natural numbers. Let ϕ_y be defined as the program encoded by the natural number y. We assume $\phi_0, \phi_1, \phi_2, \cdots$ is the list of all programs, where ϕ_0 is the special empty program. A partial function $f : N \rightarrow N$ is the function whose domain and range are subsets of N. A partial function f is computable if there is program $\phi(x)$ such that $\phi(x) = f(x)$ when $f(x)$ is defined, and $\phi(x)$ does not stop otherwise. The Halting problem is to determine if a program ϕ_x stops. This is a famous uncomputable problem (see [8]). A set $A \subseteq N$ is recursive if its characteristic function

$$\chi(x) = \begin{cases} 1 & x \in A \\ 0 & \text{otherwise} \end{cases}$$

is computable.

Theorem: Let $g(x) : N \rightarrow N$ be a partial computable function such that $g(x)$ is defined on at least one element

on N. There exists a program $p(x, y)$ with inputs on $N \times N$ such that $p(x, y) = g(x)$ for infinite many y's; and the set $\{y \in N | p(x, y) = g(x)\}$ is not recursive.

Proof: We will reduce the Halting problem to this problem. Define the function $h(x, y) = g(x)$ if ϕ_y stops in finite number of steps, and $h(x, y)$ is undefined otherwise. It is easy to see that $h(x, y)$ is a computable function. For a fixed $y \in N$, ϕ_y stops if and only if $h(x, y) = g(x)$ for all $x \in N$ if and only if $y \in \{y \in N | p(x, y) = g(x)\}$. Since the Halting problem is uncomputable, the set $\{y \in N | p(x, y) = g(x)\}$ is also not recursive.

The theorem above shows that it is undecidable to check if a constant y to make a program $p(x, y)$ to compute $g(x)$ even $p(x, y)$ computes $g(x)$ for many constants y.

8. CONCLUSIONS

We present two methods for preventing the software tempering via password. One is based on the substitution via hashing function. The second is based on the permutation via hashing function. They can be combined into one method. The methods can be broken if the password is released. It seems that using the help from hardware can improve the security of software protection, which does not involve password.

9. ADDITIONAL AUTHORS

Additional authors: Adbo Husseiny(Technology International of Virginia, 429 West Airline Highway, Suite S, LaPlace, LA 70068, email: dr_abdo@rtconline.com).

10. REFERENCES

[1] T. Aura and D. Gollman, Software license management with smart cards, Proceedings of the Usenix workshop on smartcard technology (Smartcard'99), 1999, pp. 75-86.

[2] D. Aucsmith, Tamper resistant software: An implementation. In R. J. Anderson ed., Information Hiding: First international workshop, Lecture Notes in Computer Science, V. 1174, 317-333. Springer, 1996.

[3] B. Barak, O. Goldreich, R. Impagliazzo, and S. Rudich, On the impossibility of obfuscating programs, In Proceedings of International association for crytographic research, CRYPTAO'01, 2001, LNCS 2139, pp. 1-18.

[4] S.Y. Bennet, Using secure coprocessors, Ph.D thesis, CMU-CS-94-149, Carnegie Mellon University, 1994.

[5] C. Collberg, C. Thomborson, and D. Low, A taxonomy of obfuscating transformation, Technical report 148, Department of computer science, the University of Auckland, Auckland, New Zealand, 1997.

[6] C. Collberg, and C. Thomborson, Software watermarking: models and dynamic embeddings, Proceedings of POPL'99- 26th ACM symposium on principles of programming languages, 1999.

[7] C. Collberg, and C. Thomborson, Watermarking, temper-proofing, and obfuscation - tools for software protection, IEEE traction on software engineering 8(28), 2002, pp. 735-746.

[8] N. Cutland, Computability, an introduction to recursive function theory, Cambridge University Press, 1980.

[9] A. Herzberg and S. S. Pinter, Public protection of software, ACM transaction on computer systems, 5(4) 1987, pp. 371-393.

[10] F. Hohl, Time limited blackbox security: Protecting mobile agents from malicious hosts. In G. Vigna Ed., Mobile Agents Security, Lecture Notes in Computer Science, V. 1419, Springer 1998, pp. 92-113.

[11] T. Iwai, K. Kuriyama, W. Wu, and F. Mizoguchi, A proposal for tamper-resistant mobile agents, In Computer Security Symposium (CSS99), pp. 43-48, Oct., 1999.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, Architectural support for copy and tamper resistant software, In architectural support for programming languages and operating systems, Nov. 2000, pp. 168-177.

[13] M. Mambo, T. Murayama, and E. Okamoto, A tentative approach to constructing temper-resistant software. In New Security Paradigm Workshop, pp. 23-33, Sept. 1997.

[14] A. Mana and E. Pimentel, An efficient software protection shceme, Proceedings of the 16th international conference on Information security 2001, pp.385-401.

[15] T. Maude and D.Maude, Hardware protection again software piracy, Communication of ACM, 9(27), 1984, pp. 950-959.

[16] M. Misawa, K. Akai, and T. Matsumoto, Evaluation of obfuscator by searching runtime data, In Symposium on Cryptography and Information Security (SCIS 2003), pp. 365-370.

[17] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, Software obfuscation on a theoretical basis and its implementation, IEEE Trans. Fundamentals. E86-A(1). Jan. 2003.

[18] D. Safford, Take control of TCPA, Linux Journal, August 2003.

[19] C. Wang, J. Hill, J. Knight and J. Davidson, Software tamper resistance: Obstructing static analysis of programs. Technical report CS-2000-12, Department of Computer Science, University of Virginia, Dec. 2000.

11. APPENDIX 1: AN EXAMPLE FOR PUTTING THEM TOGETHER

In the example below. The password $p = 2357$, which has $hash\_value(p) = 5392$, $hash\_value(p+1) = 5$, $hash\_value(p+2) = 3$, $hash\_address(p+1) = 4$, $hash\_address(p+2) = 1$. This example combines the method 1 and method 2. It also has the consideration to recover from the wrong password.

```
#include <stdio.h>
#include <math.h>
#include "hash.h"
#define array_size 64
#define hp 5392
double c1,c2;
void quadratic(double b, double c, double *root1, double
*root2){
double temp;
temp=sqrt(b*b-c2*c);
*root1=(-b+temp)/c1;
*root2=(-b-temp)/c1;
```

```
}
int main(){
double b,c,root1, root2;
double constants[array_size]={
3.12, 1.0, 5.12, 4.13, −3.0, 5.16, 2.17, 3.0
7.52, 6.9, 8.73, 9.23, 9.0, 8.42, 7.29, 5.9,
1.92, 9.2, 3.92, 6.63, 8.7, 8.36, 9.15, 6.2,
4.91, 4.9, 7.19, 2.76, 5.8, 8.79, 5.32, 4.9,
9.30, 2.9, 8.17, 9.26, 7.2, 3.12, 3.56, 3.7,
7.98, 6.8, 3.32, 5.78, 4.6, 1.26, 4.32, 2.8,
3.10, 5.3, 3.83, 4.28, 7.9, 3.64, 4.57, 4.9,
2.23, 3.8, 3.87, 6.12, 4.5, 4.98, 1.05, 9.0};
int i;
int password;
for (i=0; i<3; i++){ // Check the password.
scanf("%d", &password);
if(hp! = hash_value(password)) {
if (i<2) {printf("incorrect password, type again");}
else {return -1;};
}
else break;
};
c1=constants[hash_address(password + 1)] +
hash_value(password + 1); // Recover c1.
c2=constants[hash_address(password + 2)] +
hash_value(password + 2); // Recover c2.
scanf("%lf",&b);
scanf("%lf",&c);
quadratic(b,c, &root1, &root2);
printf("%lf, %lf", root1, root2);
return 0;
}
```

12. APPENDIX 2: FUNCTION POINTER ARRAY

```
#include <stdio.h>
#include <math.h>
#define c0 0
#define c1 1
#define c2 2
#define c3 3
double temp; int (*a[4])();
double b,c,root1, root2;
int step0( ){temp=sqrt(b*b-4.0*c); return 0; }
int step1( ){root1=(-b+temp)/2.0; return 0; }
int step2( ){root2=(-b-temp)/2.0; return 0; }
int quadratic( ){a[c0](); a[c1](); a[c2](); return 0; }
int main(){
a[0]=step0;
a[1]=step1;
a[2]=step2;
a[3]=quadratic;
scanf("%lf",&b);
scanf("%lf",&c);
a[c3]();
printf("%lf, %lf",root1, root2);
return 0;
}
```

13. APPENDIX 3: OBFUSCATION VIA FUNCTION POINTER ARRAY

The correct password is $p = 2387$. The hashing function hash() has the values hash(p)=7, hash(p+1)=4, hash(p+2)=10, hash(p+3)=2, hash(p+4)=11, and hash(p+5)=13. If the correct password is provided by the user in the beginning of the program, the constants c0, c1, c2, c3, c4, c5 will get the correct values 0, 1, 2, 3, 4.0 and 2.0, respectively.

```
#include <stdio.h>
#include <math.h>
#include "hash.h"
int c0, c1, c2, c3;
double c4, c5;
double temp;
int (*a[4])();
double b,c,root1, root2;
int step0( ){temp=sqrt(b*b-c4*c); return 0;}
int step1( ){root1=(-b+temp)/c5; return 0;}
int step2( ){root2=(-b-temp)/c5; return 0;}
int quadratic( ){a[c0](); a[c1](); a[c2](); return 0;}
int main(){
a[0]=step0; a[1]=step1;
a[2]=step2; a[3]=quadratic;
scanf("%d",&passwrod);
c0=-7+hash(password);
c1=-3+hash(password+1);
c2=-8+hash(password+2);
c3=1+hash(password+3);
c4=-7.0+hash(password+4);
c5=-11.0+hash(password+5);
scanf("%lf",&b); scanf("%lf",&c);
a[c3]();
printf("%lf, %lf",root1, root2);
return 0;
}
```