Protecting Secret Keys in a Compromised Computational System

W. P. R. Mitchell
Department of Computer Science
University of Manchester
Manchester M13 9PL
bill@cs.man.ac.uk

August 9, 1999

Abstract

Software applications which run in a compromised environment and perform sensitive operations, such as providing a secure communication channel between two individuals, require protection in order to prevent them being run by an unauthorised adversary.

This paper looks at how to build in some protection against an adversary who wishes to modify an application so that it no longer authenticates the user before running. This protection works against a casual observer, that is someone who has access only to standard debugging tools, visualisation tools, and decompilers. The tricks given in the paper do not work against an all powerful adversary.

The paper treats the problem of protecting the code fragments dealing with authentication as equivalent to encrypting plaintext without revealing the secret key.

1 Introduction

Consider a software application which is only intended to be used by a specific individual, or within a specified location, or allows privileged access to data by users depending on their role. For example the application may allow one person to secretly communicate with another person over a secure channel. Or the software is a commercial product which is licensed to only run for one person located at any machine in a particular network. Another example is a database which allows some users to have certain privileged access to the data and other users only normal access to the data. It may be perfectly allowable for an individual to check that their own company records are correct, but not to access the personal records of any other individual.

The most obvious way to prevent an adversary from using a piece of software without permission from the rightful user is for the owner to keep it encrypted and only decrypt it when it is being used. However this will not work in a situation where the software is decrypted by a supervisor, but used by a subordinate who may decide to supply a copy of the code to an adversary, or the software is used in an open network where potentially anyone could copy the unencrypted code whilst it is in use. There is then a real need to ensure that the application contains mechanisms which ensure it can not work without proper permission to do so.

The application will typically be protected from use by anyone but the rightful owner by means of some kind of key, perhaps a hardware key such as a dongle, perhaps by a smart card, perhaps by communication with a network license manager, or by some password which only the user knows. However the application must contain some secret information which it uses during the authentication stage to validate the key. This is a possible point of attack if a copy of the software falls into the wrong hands.

If an attacker can reverse engineer the authentication process they may then be able to bypass this authentication process altogether. In the case of the communication software the attacker will then be able to impersonate the rightful user, in the case of the software pirate they can then make hacked copies of the application and give them away. Reverse engineering of binary code is a real danger for secure applications, [**3**] [**4**], [**5**], [**6**] give details of sophisticated decompilation tools and decompilers currently available or under development.

We will regard the problem of preventing the authentication process from being subverted as equivalent to computing the encryption of a piece of plaintext without revealing the encryption key. This paper considers the problem of how to compute the encryption in such a way that the key remains hidden from a casual observer. By this we mean an observer who can only analyse the system through use of standard debugging tools, visualisation tools, and decompilers. It does not include the case of an all powerful observer who can apply any arbitrary technique in analysing the system.

There are secondary problems in protecting an authentication process which we also consider. An adversary may decide that rather than spend time and energy reverse engineering the authentication process they will simply rip it out from the code altogether. This is sometimes possible if the authentication is implemented in a naive way. The protection mechanisms have to be incorporated so they can not be removed by blind attacks which require no understanding of how the code works.

A challenge response authentication often involves the use of a dongle. Checksums of various code fragments are commonly used to decide if the code has been tampered with. There are standard blind attacks against both of these methods which we look at in sections 2 and 3. Finally in section 4 we look at how to hide a secret key whilst encrypting some plaintext with the key.

The protection tricks discussed here are designed to be immune to blind attacks. They are designed to force an attacker to understand how the code works, rather than simply identify vulnerable areas of code which can be bypassed, using standard observation tools. The tricks are also designed so that it is hard to break them down into smaller parts which can be individually attacked through dynamic slicing techniques [**7**], [**10**]. This forces the attacker to understand the dynamics of the whole protection mechanism during run time in order to attack any part of it. The tricks are designed to be randomly varying throughout their structure, in essence this makes them appear as self modifying code which operates in an arbitrary fashion. This prevents any standard decompiler from reconstructing any meaningful high level source code.

The tricks involve representing the keys as abstract data types, whose structure is dynamically varied during run time. The same is true for the encryption algorithm. Both the keys and the encryption algorithm are represented through the use of automaton with certain additional features. All of the important encryption operations are translated into operators on automaton which makes it difficult to distinguish operations during run time. The structure of all the operations also dynamically varies during run time.

In order to uncover the computation an attacker will be forced to uncover the semantics of the tricks which is beyond a casual observer. Using that semantic knowledge the attacker then has to identify which parts of the code implement the different semantic operations and data types. Even if the attacker is given this document to work from, assuming that the application is written in some declarative language which is optimised by the compiler, simply using standard observational tools will not enable them to uncover the keys. This is true since recovering the secret key is equivalent to using a decompiler to reconstruct a universal abstract data type interpreter from the static machine code.

1.1 Related Work

One of the most interesting ideas for protecting an application is to build self encrypting code [**1**]. The idea is that each block of machine code is a decryption cipher, which decrypts the next block of code to run, plus some small part of the application which is encrypted. After running the block re-encrypts itself. This method is a good way to bolt on security after the application is built. It has implications for maintenance since bugs may subtly interact with the modified machine code. It also has portability costs. Finally should an adversary suspect that the code is decrypting itself one block at a time it will then be easy to recapture a fully working version of the unprotected code.

An approach taken in such products as [**2**] is to build a compiler which uses novel optimising techniques which are not easily reversible, so effectively hiding the authentication process. This has portability issues and is more difficult to maintain than an abstract protection mechanism built into the code itself. The other drawback is that the final code has static protection mechanisms built into it, which by definition are simpler to crack than dynamically varying ones.

Using the idea of hiding values in a higher order abstract data type is not new, in [**8**], [**9**], [**12**] they show how to protect against reverse engineering by such tricks. Their work is concerned with allowing an algorithm to run in such a way that the adversary can not reconstruct the original algorithm, thus protecting the intellectual property contained in the algorithm. This is different from our problem, where we are not concerned with the algorithm being discovered, but with the more difficult problem of hiding some of the values which pass through the algorithm.

In [**11**] they look at the related problem of how to use a personal key to designate authority to temporary subkeys in a compromised system without revealing too much about the personal key. That paper is not concerned with the actual mechanics of preventing reverse engineering of the authentication process, rather it looks at ways of sharing out some of the attributes attached to the main key without allowing an unacceptable amount of information to leak.

2 Bat Attacks

This section deals with the problem of protecting the application with a hardware key which is only used as part of a simple challenge response mechanism. The response to the challenge is incorporated into a boolean valued test which determines whether the application should proceed or take evasive action.

The most common form of hardware key is a dongle so we will use this term to refer to a generic hardware key from now on. A simple challenge response test using a dongle may look like:

```
if (QuerryDongle(Challenge Value) == Right Response) {
    Continue as normal;
   } else {
    Take evasive action;
   }
```

One weakness with this kind of test is that `QuerryDongle` typically uses a fixed port to communicate whose address is hardwired into the dongle. Current debugging tools can insert break points which occur when that port is used. This effectively locates the point in the code where the boolean test occurs. An attacker can now replace

```
(QuerryDongle(Challenge Value) == Right Response)
```

with `True`. Dongle manufacturers commonly warn about this type of attack in their literature.

We call an attack where certain boolean tests are replaced by a constant value (true or false) a **boolean adaptation of tests** attack (**bat** attack for short, or sometimes boolean attack).

2.1 bat defense

Replace the above simple test with a finite graph G where each node is labeled by a boolean test of the form

`(QuerryDongle(`v`) == `r`)`

and each node has two edges leading from it, one labeled 0 the other labeled 1. The r values are constant and vary from state to state, v is a variable. Choose G in an arbitrary manner. Fix one of the nodes s as the start node.

A sequence of values v_i generates a bit string b as follows. Define states $s_i \in G$ and $b_i \in \mathbb{B}$ inductively, where node s_i is labeled by `(QuerryDongle(`v`) == `r_i`)`:

- $s_0 = s$
- $b_i =$ `(QuerryDongle(`v_i`) == `r_i`)`

  s_{i+1} is the state for which there is an edge $s_i \xrightarrow{b_i} s_{i+1}$.

Assuming that there are no terminating nodes in G, any sequence of values v_i generates a bit string $b_0b_1b_2\cdots$. Let this bit string be denoted $\mu(G, v_0v_1\cdots)$

The application incorporates the graph G and a predetermined set of values v_i together with the precalculated value $\alpha = \mu(G, v_0v_1\cdots)$. Before the application runs (or periodically whilst running) it runs through G with values v_i recalculating dynamically $\alpha' = \mu(G, v_0v_1\cdots)$. At various places in the code, dislocated from the immediate calculation of α', the application checks $\alpha == \alpha'$ and only now chooses to continue or take evasive action.

The boolean tests in G can be located in the application code by inserting relevant break points which monitor port traffic. If an attacker wishes to replace any of the boolean tests in G with a constant value then they have a 1/2 probability of choosing the correct value so that α' will still be α. If there are n values v_i the attacker has only a $1/2^n$ probability of correctly replacing each boolean test without affecting α'. In fact the probability is less than this since there can be many more states in G than n.

This defense has a weakness in that the attacker may discover where the new boolean tests $\alpha == \alpha'$ are and simply replace them with `True`. This attack is much harder since there is no longer any port communication to give the game away. It is possible to disguise this comparison by hiding α and α' in higher order data types, and then comparing them in a deceitful manner. This is a special case of the next section.

3 Lute Attacks

It is quite fashionable to try to protect applications by calculating checksums on those parts of the code which authenticate the user to ensure that the code has not been tampered with. Some hash function or message authentication checksum (MAC) is used on a part of the code dealing with the authentication process. The resultant value is built into another part of the code. When the application runs it recalculates the relevant hash value whenever necessary to check it is still the same.

If the hash values are used in a simple boolean test they are vulnerable to bat attacks. In order to calculate a hash value of a code fragment the application will need to access some storage device (e.g the disk drive). Such access can be monitored and so break points can be used in the same way as in the dongle case above to locate the boolean tests.

Rather than use a hash value in a boolean test it is sometimes incorporated into a calculation in the application itself. Here is a trivial example which is not particularly secure. If file `f` has hash value `a` which is stored as a constant somewhere and `h` is the hash function, then an assignment `x = b` can be replaced with `x = b - a + h(f)`. Tampering with `f` will cause this to give the wrong value to `x`.

The most obvious attack here is to insert breakpoints on `h` (assuming this can be achieved by tracking when IO occurs on binary files). From these we can determine what the constants `a` are and place them in a look up table. Then change the code for `h` so that it uses the look up table of the values `a` and simply returns the right value. This is much more work than a bat attack, but still doable. We call such an attack a look up table evaluation attack, or a lute attack for short.

For the rest of this section we will look at one way of representing a constant in terms of a randomly varying abstract data type, and how to defend against a lute attack when performing a checksum on a code fragment.

3.1 Hiding a constant value in an automaton

Let A be a nondeterministic finite state automata with states S and labeled edges E, with labels taken from the set L. Let s_0 be the start state. Choose a function $\delta : \mathbb{N} \longrightarrow S$.

A sequence $l = l_0 l_1 \cdots l_n \in L^*$ determines a value in $\mathbb{B}^N$ as follows. Let $A(l)$ be the set of states which are reached on input l to A. Choose $\beta \in \mathbb{B}^N$ by

$$\beta_i = 1 \Leftrightarrow \delta(i) \in A(l)$$

This provides a mapping from L^*, A and a mapping $\delta : \mathbb{N} \longrightarrow S$, to $\mathbb{B}^N$. Let $\omega(A, l, \delta)$ denote this mapping.

Next consider how to construct an A with string $l \in L^*$ and subset $U = A(l)$, which represents a given value $\beta \in \mathbb{B}^N$. Let $l = l_0 l_1 \cdots l_n \in L^*$. Construct at random a directed acyclic graph (dag) with a single root, using U as the set of leaf nodes, where each branch has length n, and label each edge in the branch with the relevant l_i so that each branch is annotated with l.

Now add edges and states at random (the result is no longer required to be a dag), but in such a way that no new paths are created which have prefix l at the root. The set U now consists of those states which are reached on input l to A, i.e. $U = A(l)$. Choose functions $\delta_1 \{i \in \mathbb{N} \mid \beta_i = 1\} \longrightarrow A(l)$ and $\delta_2 \{i \in \mathbb{N} \mid \beta_i = 0\} \longrightarrow S - A(l)$, putting these together gives a function δ where

$$\delta(i) \in A(l) \Leftrightarrow \beta_i = 1$$

For the given A and l and map δ, $\omega(A, l, \delta) = \beta$. Notice that to a large extent the construction of l, A and δ is completely random. This allows the value β to be dynamically stored in a randomly varying way. The size of A and l can be made arbitrarily large, depending on how much computation is acceptable for the recovery of a constant value.

Any one attempting to recover β would have to locate the map δ, and recover the semantics behind the construction of A. This will be impossible for a casual observer. Notice that it is possible to recover any single bit of β without revealing any of the other bits. To find β_i, recover the set U from the automata and check if $\delta(i) \in U$.

3.2 Varying the automaton representation of a bit string

This section briefly outlines ways to construct new automaton representations of a bit string β from old ones without revealing β and in an efficient manner. One of the goals of using an abstract data type to represent a constant value is to be able to vary the representation dynamically in a random fashion and without revealing the constant. The automaton representation is quite effective at doing this.

A bit string can be hidden by a triple (A, l, δ) as described above. There are simple ways which can be used to change each item in this triple while keeping $\omega(A, l, \delta)$ invariant.

Let $U = A(l)$. Using new states not in A construct a new dag D whose leaves are given by U and where every branch from the root s'_0 to a leaf $u \in U$ is annotated by the same word $l' \in L$. Add new nodes and edges at will to D so long as no new paths are constructed which are prefixed by l'. Let A' be the automaton consisting of the union of D with any random subautomaton chosen from A. Take s'_0 as the start node for A'. Modify the map δ so that any values $\delta(i)$ which are no longer defined are reassigned to states not in U, let δ' be the resultant map. Notice that

$$\omega(A, l, \delta) = \omega(A', l', \delta')$$

The only unchanged quantity between (A, l, δ) and (A', l', δ') is the set $A(l) = A'(l)$.

The triple (A, l, δ) can also be changed by applying the following graph rewriting rule. For any node u we can add a new node u' so long as we also add the following edges.

- If there is an edge $w \xrightarrow{a} u$ add an edge $w \xrightarrow{a} u'$
- If there is an edge $u \xrightarrow{a} w$ add an edge $u' \xrightarrow{a} w$

Note that if $u \in A(l)$ then we will also now have $u' \in A(l)$. We may also delete any subgraphs which are not path connected to any node of $A(l)$. One may even delete some, but not all, of the nodes in $A(l)$ providing that suitable modification is made to δ so that every undefined value is mapped to some other value in $A(l)$.

The map δ can be modified to a large extent without affecting $\omega(A, l, \delta)$. The only attribute that matters for the map δ is that it maps the correct values into $A(l)$. Thus we may compose δ with any function $\nu : S \longrightarrow S$ so long as $\nu(A(l)) \subseteq A(l)$ and $\nu(S - A(l)) \subseteq S - A(l)$.

Possibly the easiest and most useful modification of (A, l, δ) is to apply a random permutation of the node set. Let $A = (S, E, s_0)$ be an automaton with states S, edges E and where s_0 is the start state. For a permutation $\rho : S \longrightarrow S$ we can define $\rho(A) = (S, \rho(E), \rho(s_0))$, where

$$\rho(u \xrightarrow{l} v) = \rho(u) \xrightarrow{l} \rho(v)$$

Define $\rho(A, l, \delta) = (\rho(A), l, \rho \circ \delta)$. For any $l \in L^*$,

$$\omega(A, l, \delta) = \omega \circ \rho(A, l, \delta)$$

This works since $(\rho \circ \delta)^{-1} \rho(A)(l) = \delta^{-1} A(l)$. Also notice that given $(A', l, \delta') = \rho(A, l, \delta)$ and no knowledge of ρ, it is an NP complete problem to show that (A', l, δ') is isomorphic to (A, l, δ).

Using various combinations of the above modifications can alter the original triple (A, l, δ) into a radically different structure. The modifications are elementary and therefore efficient (in terms of the size of A). This makes it a straightforward matter to dynamically alter the structure representing β in a virtually arbitrary way without revealing β during the modification.

3.3 lute defense

Let $T(\beta, r)$ be a function which returns a triple (A_r, l_r, δ_r) where $\omega(A_r, l_r, \delta_r) = \beta$. Function T uses some pseudo-random number generator with seed r to pseudo-randomly construct (A_r, l_r, δ_r) in some way.

The hash function $\mathtt{h(f)} \in \mathbb{N}$ is replaced by a function $\mathtt{H(f, r)}$. This returns the triple $T(\mathtt{h(f)}, r) = (A_r, l_r, \delta_r)$ where $\omega(A_r, l_r, \delta_r) = \mathtt{h(f)}$. It should be noted that if $\mathtt{H(f, r)}$ is merely a call to $T(\mathtt{h(f)}, r)$ then this will not be secure, since the secondary call to $\mathtt{h(f)}$ is still susceptible to a lute attack. The function $\mathtt{H(f, r)}$ must build a suitable triple by incorporating the value of $\mathtt{h(f)}$ directly into the construction. The checksum constants $\mathtt{a}$ are replaced by $T(\mathtt{a}, \mathtt{r})$.

The assignment `x = b` could now be replaced by the following trick pseudo-code. In the code we assume that triples (A, l, δ) are defined as some suitable structure in C++ so that if `p` is a pointer to a triple (A, l, δ), then `p->A` $= A$ etc.

```
\* if the code is unmodified then h(f) == a *\
r = random number;
\\ * is the dereference operator
*p = T(a, r);
ρ = random permutation of states of p->A;
*q = ρH(f, r);
n = random number;
ρ = random permutation of [1,2,..., n];
u = [p, q];
\* supposedly *p is equivalent to *q *\
int y[];
int z[] = random partition of b into n integers;
int e[];
x = 0;
for (i=1; i <= n; i++) {
    e[i] = random choice of -1 or 1;
    y[i] = random element of L*;
};
for (i=1; i <= n; i++) {
    v = random choice of 1 or 0;
    \\ note here * is multiplication
    x += e[i] * ω(u[v]->A, y[i], u[v]->δ) + z[i];
};
for (i=1; i <= n; i++) {
    v = random choice of 1 or 0;
    x -= e[ρ(i)] * ω(u[v]->A, y[ρ(i)], u[v]->δ);
};
\* now x == b *\
```

A casual observer will not be able to replace `H(f,r)` by a constant value without destroying the final value of `x`. The values `*p` and `*q` are equivalent in the sense that for all values $x \in L^*$,

$$\omega(\mathtt{p->A},\ x,\ \mathtt{p->}\delta) = \omega(\mathtt{q->A},\ x,\ \mathtt{q->}\delta)$$

For the adversary to realise that `p` is just a permutation of `q` they would need to recalculate the first value of ρ which is an NP complete problem. Of course if the adversary realises that ρ is computed in the code they could just store it somewhere for later use.

However a casual observer will not be able to understand the purpose of ρ or even that it is a permutation of states in an automaton.

The next section turns to the major problem of how to compute the encryption of some plaintext without revealing the key.

4 Hidden Key Computation

If an adversary can not bypass the authentication code contained in the application an alternative is to uncover any secret keys used in the authentication process so that they can use them in a man in the middle attack to fool the software into running. For this section we will restrict our attention to the problem of how to encrypt some plaintext without revealing the secret key used by the encryption cipher. Before we go into this problem in detail we first have to solve the problem of choosing elements of a list in such a manner that the indices of those elements remain hidden. This will be a useful tool which we use when solving the main problem.

4.1 Choosing elements from a list

This section considers how to choose a subsequence from a list of bits in a manner which hides the locations of the chosen bits. Let $[b_0, \ldots, b_n]$ be a list of bits. The problem is how to construct a subsequence $[b_{i_0}, \ldots, b_{i_m}]$ without revealing the values i_j to a casual observer. In order to simplify the situation only the case of choosing a single bit b_j is considered.

Let $\mathcal{P}_n = \{p_1, \ldots, p_n\}$ be a set of n propositions. Let $\mathbb{B}\mathcal{P}_n$ be the set of propositional formulae constructed from $\mathcal{P}_n$. Let $\beta = [b_0, \ldots, b_n]$, let $\theta : \mathcal{P}_n \longrightarrow \mathbb{B}$ be the valuation given by $\theta(p_i) = b_i$. This extends to a valuation on propositional formulae as usual.

For a set X let $\mathsf{rand}\, X$ denote an element of X chosen at random. Each occurrence of $\mathsf{rand}\, X$ in an expression denotes a separate instance of such a random choice, implying that the value of $\mathsf{rand}\, X$ varies at each occurrence.

For a propositional formula f let $[f]$ denote the set of formulae which can be derived from f by a standard predetermined set of equivalences. For example the set of equivalences might be

$$\begin{array}{lcllcl}
x \wedge (y \vee z) & \cong & x \wedge y \vee x \wedge z & x \vee (y \wedge z) & \cong & (x \vee y) \wedge (x \vee z) \\
\neg(x \wedge y) & \cong & \neg x \vee \neg y & \neg(x \vee y) & \cong & \neg x \wedge \neg y \\
x \rightarrow (y \rightarrow z) & \cong & (x \wedge y) \rightarrow z & x \rightarrow y & \cong & \neg x \vee y \\
x \vee (y \vee z) & \cong & (x \vee y) \vee z & x \wedge (y \wedge z) & \cong & (x \wedge y) \wedge z \\
x \wedge y & \cong & y \wedge x & x \vee y & \cong & y \vee x
\end{array}$$

Then, for example, $p \rightarrow (\neg q \rightarrow r) \ \in \ [p \rightarrow (q \vee r)]$.

We can construct random tautologies over $\mathbb{B}\mathcal{P}_n$ as follows. Pick $f = \mathsf{rand}\, \mathbb{B}\mathcal{P}_n$, define the tautology τ as

$$\tau = \mathsf{rand}\, [(\mathsf{rand}\, [\neg f]) \vee (\mathsf{rand}\, [f])]$$

Similarly we can define a random falsehood ∂ (a formula which always evaluates to false) as

$$\partial = \mathsf{rand}\, [\neg((\mathsf{rand}\, [\neg f]) \vee (\mathsf{rand}\, [f]))]$$

Choosing $\beta[i]$ deceitfully

Let P be the set of permutations of $[1, 2, \ldots, n]$. Such a permutation extends to $\mathbb{B}\mathcal{P}_n$ in the usual way. That is for $\rho \in P$, $\rho(f)$ is the result of replacing each occurrence of p_i by $p_{\rho(i)}$. Choose a random permutation p of $[b_0, \ldots, b_n]$ such that, if $[a_0, \ldots, a_n] = p([b_0, \ldots, b_n])$, then $a_n = b_j$.

The following code fragment, written in C++ pseudo-code results in the variable x having the value b_j at the termination of the second for loop. A casual observer will not be able to recover the value j even though they can observe the value x.

```
τ[0] = τ;
∂[0] = ∂;
for (int i = 1; i <= n; i++) {
    ρ = rand P ;
    τ[i] = rand[ρ(τ[i-])];
    ∂[i] = rand[ρ(∂[i-])];
}
x = 0;
for (int i = 0; i <= n; i++) {
    d = disjunctive normal form of a_i ∧ τ[n-i]∨∂[n-i]∧x;
    x = θ(d);
    Phony = Phony ∧x;
};
```

Note that after this code fragment has executed

$$x = b_j \wedge 1 \vee 0 = b_j$$

The sequence τ[i], and ∂[i] are designed to hide the fact that τ and ∂ are only intended to be used for extracting b_j. It would seem that τ and ∂ construct some meaningful sequence which is in fact arbitrary. Making the sequence random ensures that the construction dynamically changes for every instance of finding b_j. Introducing the random permutation ρ means that τ[i] and ∂[i] may well no longer be either a tautology or a falsehood. This implies that any correlation between x and the values a_i is broken.

All the bits b_i are used in the loop, nothing distinguishes one from the other. Only the permutation p distinguishes i. The place where p is used is dislocated from the use of the values a_i which hides the special nature of i.

If the value of x is traced it will take a random sequence of values, which is different on each run, with the exception of the final value which will always be b_j. This could also be hidden by using a two element array $X = [x_0, x_1]$. Replace the second for loop above with this one

```
int r;
int X[2];
for (int i = 0; i <= n; i++) {
    r = rand {0, 1} ;
    d = disjunctive normal form of a_i ∧ τ[n-i]∨∂[n-i]∧X[r];
    X[r] = θ(d);
    Phony = Phony ∧X[r];
};
```

Now the value b_j is given by the value $X[r]$ at the end of the for loop. Thus for any particular run b_j might be the value of x_0 or of x_1. Both these values will vary at random over a series of program runs. An observer will have to know which of x_0 or x_1 holds b_j. To make this even more confusing any size array could be used for X. Using a two element array would seem a reasonable choice since that is the minimum number required to ensure there is no statistical relationship between the values in the array and $b_j = a_n$.

4.2 Deceitful automata

This section discusses how to perform the encryption $\psi_s(k_s, a)$ of text a with cipher ψ_s and key k_s while keeping the key hidden from a casual observer. The proposed method is based on the idea of constructing a graph which represents several parallel encryptions whose intermediate values are all permuted together in seemingly random fashion into a single bit string. This creates the impression that only a single complex encryption is being computed and effectively hides which bits belong to the real encryption cipher.

One of the computations represents $\psi_s(k_s, a)$. The graph has the property that any path of the right length can be used to construct $\psi_s(k_s, a)$ while keeping k_s hidden. The graph is dynamically constructed during run time and will change at random from one execution of the program to the next.

Assume the encryption cipher is an r-round cipher [**13**] where each round is given by the function $\chi(k^{(i)}, a_i)$, where $k^{(i)}$ is the key material used at the i-th round from the key k, and a_i is the ciphertext of the i-th round. That is the encryption $\psi_s(k, a)$ of plaintext a with key k is given by the iteration

$$\chi(k_r, \chi(k_{r-1}, \ldots, \chi(k_0, a)\cdots)$$

Suppose that the cipher encrypts blocks of size n.

Not all encryption ciphers fit this description, for example the Twofish cipher [**14**] doesn't. However block ciphers usually include many rounds using essentially the same function, and this part of the cipher can be protected using the ideas in this section.

An extraction function $\varepsilon : \mathbb{B}^{ln} \longrightarrow \mathbb{B}^n$ is any function which returns a permuted subsequence of its argument. An extraction function can be completely specified by a sequence $(i_1, \ldots i_n)$, where

$$\varepsilon(b_1 \cdots b_{ln}) = b_{i_1} \cdots b_{i_n}$$

A block permutation function $\sigma : (\mathbb{B}^n)^l \longrightarrow \mathbb{B}^{ln}$ is any function which can be written as

$$\sigma(x_1, \ldots, x_l) = \rho(x_1 @ \cdots @ x_n)$$

where $\rho : \mathbb{B}^{ln} \longrightarrow \mathbb{B}^{ln}$ is a permutation and @ is append. That is σ appends its arguments together and then permutes the resultant bit string.

A muddled encryption graph is a finite state graph with nodes labeled by l-tuples $(\langle\varepsilon_1, \chi_1\rangle, \ldots, \langle\varepsilon_l, \chi_l\rangle)$, and edges labeled by block permutations. Each ε_i is an extraction function and χ_i is a round-function of an encryption cipher.

Suppose that

$$(\langle\varepsilon_1, \chi_1\rangle, \ldots, \langle\varepsilon_l, \chi_l\rangle) \xrightarrow{\sigma} (\langle\varepsilon'_1, \chi'_1\rangle, \ldots, \langle\varepsilon'_l, \chi'_l\rangle)$$

is an edge A of a muddled graph. The edge A is dovetailed if there is a permutation $p : \mathbb{B}^n \longrightarrow \mathbb{B}^n$ such that for every $(e_1, \ldots, e_l) \in (\mathbb{B}^n)^l$

$$\varepsilon'_i(\sigma(e_1, \ldots, e_l)) = e_{p(i)} \text{ and } \chi'_i = \chi_{p(i)}.$$

A node $(\langle\varepsilon_1,\chi_1\rangle,\ldots,\langle\varepsilon_l,\chi_l\rangle)$ is interleaving if for all $\alpha \in \mathbb{B}^{ln}$, $\varepsilon_1(\alpha)@\cdots@\varepsilon_l((\alpha)$ is a permutation of $(\alpha$. Define a muddled graph where all the edges are dovetailed and all the nodes are interleaving to be a deceitful cipher automaton.

For a node $\eta = (\langle\varepsilon_1,\chi_1\rangle,\ldots,\langle\varepsilon_l,\chi_l\rangle)$, $\alpha \in \mathbb{B}^{ln}$, and a sequence of keys $k = (k_1,\ldots k_n)$ write $\eta(k,\alpha)^{(u)}$ to denote the tuple

$$(\chi_1(k_1^{(u)},\varepsilon_1(\alpha)),\ldots,\chi_l(k_l^{(u)},\varepsilon_l(\alpha)))$$

Where $k_i^{(u)}$ is the key material used by the cipher χ_i in the u-th round of encryption.

Proposition 4.1
Let $k = (k_1,\ldots k_n)$ be a sequence of keys and let

$$\eta_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_r} \eta_{r+1}$$

be a path in the deceitful automaton D. Let $\eta_i = (\langle\varepsilon_1^i,\chi_1^i\rangle,\ldots,\langle\varepsilon_l^i,\chi_l^i\rangle)$. Let each χ_j^r be part of an r-round encryption cipher ψ_j as described above.

For $a \in \mathbb{B}^n$ define $\alpha_i \in \mathbb{B}^{ln}$ recursively for $0 \le i \le r$ as

$$\alpha_{i+1} = \sigma_i(\eta_i(k,\alpha_i)^{(i)})$$

and $\alpha_0 = a@\cdots@a$, then

$$\varepsilon_i^{r+1}(\alpha_r) = \psi_i(k_i,a)$$

This is a result for any path of length r in the deceitful automaton. Thus every path of length r can be used to calculate the various ψ. The result is true since the edges are all dovetailed so the χ_j^i of each node are all matched together in the α_i sequence.

The deceitful automaton D is used to deceive an attacker as follows. There are various ciphers ψ_i corresponding to the round-functions χ_i which decorate the nodes. Only one of the ψ is the real cipher, say ψ_s, the others are irrelevant. Let χ be the round-function of cipher ψ_s. It would be acceptable to choose all the ψ to be the same as ψ_s, so that every $\chi_i = \chi$. We do not do this here for reasons of generality.

Generate a random path

$$\eta_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_r} \eta_{r+1}$$

in D of length r. To perform the encryption $\psi_s(k_s,a)$ calculate the α_r as above using the key-list $k = (k_1,\ldots,k_l)$. The only key in k of interest is k_s the others are all dummies. What matters is that k_s is not discovered by the attacker. For each node η_{i+1} there is a $v \in \mathbb{N}$ such that $\varepsilon_v^{i+1}(\alpha_{i+1})$ corresponds to the value of a after i rounds of encryption with the cipher ψ_s and key k_s. That is

$$\varepsilon_v^{i+1}(\alpha_{i+1}) = \chi(k_s^{(i)},\chi(k_s^{(i-1)},\ldots,\chi(k_s^{(1)},a)\cdots)$$

Since we are placing key k_s as the s-th element of k this implies that for η_1, $v_1 = s$. The value v can be calculated afresh each time the key material is constructed for a particular path.

The key material is stored in a deceitful way in another automaton, as described earlier in section 3.1, and does not appear as a constant anywhere. The other dummy keys are derived from the same storage automaton as k_s but in a random fashion. This means none of the key material is compromised during the encryption. The final encrypted value $\psi_s(k_s,a)$ is given by $\varepsilon_{v_{r+1}}^{r+1}(\alpha_r)$.

An attacker who is only able to naively analyse the computation will not be able to learn how the deceitful automaton is used or which k_i is relevant. The deceitful automaton can be constructed dynamically and changed at random which will confuse any decompiler.

To construct a deceitful automaton generate any number of nodes as seems appropriate with randomly generated ε extraction functions in the labels, but so that each specific node is interleaving. The same set of encipher round-functions is used to construct each node. For any pair of nodes it is possible to construct a dovetailed edge between them, so assign edges between nodes totally at random. This construction allows for the deceitful automaton to be constructed in an almost completely arbitrary manner, and the total number of such automaton is huge. That makes analysing the generated automaton for patterns futile.

For an attack to succeed the attacker must realise that they need to capture the automaton and discover which ψ_i are relevant. Given that (which means capturing ε_s) they can now attempt to backtrack the calculation in the automaton. That requires capturing the relevant nodes and edges and calculating suitable inverses to the various ε and σ in order to recover all the parts of the key material used to reconstruct k_s.

Recall that an extraction function is specified as a sequence of integers. This can also be deceitfully hidden in the guise of an automaton which further complicates any attack. The techniques described in section 4.1 can be used to evaluate the final extraction function, further still hiding any starting point from which to backtrack the calculation of $\psi_s(k_s, a)$.

5 Conclusion

We have shown how to construct various abstract data types which hide secret keys used during encryption, and how to construct an apparently random dynamically varying computation which executes that encryption. The datatypes are essentially automata with some additional features. They are designed so they can be dynamically changed in an apparently random fashion without revealing the original secrets. The true nature of the data is hidden within the semantics underlying the representations which can not be recovered from the code by a casual observer.

The constructions must by their nature make the encryption cipher run less efficiently. However the cost of the extra protection is in some part made up for in their effectiveness. Even if the adversary understands the semantics of the deceitful mechanisms given here, they will still have to perform exponential searches to recover any useful material, since no standard decompiling tool will be able to reconstruct any meaningful source code. Without some sort of high level source code to work with an attacker can not make any use of their semantic knowledge. Also, the authentication process is usually performed sparingly and often is not as time critical as the rest of the application so that the extra cost may not have a great impact on performance.

Looking again at the idea of self encrypting machine code in [**1**] we can see how to effectively combine that work with the tricks given here. Instead of code which contains its own encryption key, take the key as hidden in some kind of smart card say. Parts of the application stay encrypted until decrypted using the key in the smart card, afterwards they are re-encrypted. Using the tricks here we can disguise the encryption process and hide the key during run time so that a casual observer will not be able to recover a working copy of the application. The encryption process now occurs at a higher abstract level which does not have the weakness of static machine code.

References

[1] David Aucsmith, Tamper Resistant Software: An Implementation. Proceedings of Workshop on Information Hiding, Cambridge 1996, Springer-Verlag, Lecture Notes in Computer Science, Volume 1174, pp 317–333.

[2] DashO-Pro Obfuscator optimising compiler for Java.
http://www.preemptive.com/DashO/obfuscate.html

[3] Cifuentes, Cristina and K. John Gough, Decompilation of binary programs. Software — Practice and Experience, vol. 25, no. 7, pp. 811-829, July, 1995.
http://www.csee.uq.edu.au/~cristina/pubs.html

[4] C Cifuentes, Partial Automation of an Integrated Reverse Engineering Environment of Binary Code. Proceedings Third Working Conference on Reverse Engineering. Monterey, CA, Nov 8-10 1996. IEEE-CS Press. pp 50-56.
http://www.csee.uq.edu.au/~cristina/pubs.html

[5] C Cifuentes, D Simon and A Fraboulet, Assembly to High-Level Language Translation. Proceedings of the International Conference on Software Maintenance, Washington DC, USA, Nov 1998, IEEE-CS Press, pp 228-237.
http://www.csee.uq.edu.au/~cristina/pubs.html

[6] C Cifuentes and A Fraboulet, Interprocedural Data Flow Recovery of High-level Language Code from Assembly. Technical Report 421, Department of Computer Science and Electrical Engineering, The University of Queensland, Dec 1997.
http://www.csee.uq.edu.au/~cristina/pubs.html

[7] Cimitile A., De Lucia A., Munro M., A Specification Driven Slicing Process for Identifying Reusable Functions, Journal of Software Maintenance : Research and Practice, 8(3), pp145–178, 1996

[8] Collberg, Christian, Clark Thomborson and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July, 1997.
http://www.cs.auckland.ac.nz/
~collberg/Research/Publications/CollbergThomborsonLow97a/index.html

[9] Collberg, Christian, Clark Thomborson and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. Department of Computer Science, University of Auckland, New Zealand, July, 1997.

Also in ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98), San Diego CA, January 1998.
http://www.cs.auckland.ac.nz/
~collberg/Research/Publications/CollbergThomborsonLow98a/index.html

[10] De Lucia A., Fasolino A.R., Munro M., Understanding Function Behaviors through Program Slicing, Workshop on Program Comprehension, IEEE Press, 1996.
http://www.dur.ac.uk/~dcs1elb/delucia-wpc96.html

[11] Oded Goldreich, Birgit Pfitzmann and Ronald L. Rivest, Self-Delegation with Controlled Propagation — or — What If You Loose Your Laptop, CRYPTO' 98, LNCS 1462, pp. 153-168

[12] Java Code Engineering & Reverse Engineering.
http://meurrens.ml.org/ip-Links/Java/CodeEngineering/mm_scale.html

[13] Bruce Schneier. Applied Cryptography, Second Edition, John Wiley & Sons, 1996

[14] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson. Twofish: A 128-Bit Block Cipher, June 1998,
http://www.counterpane.com/twofish-paper.html