Protecting Programs from Hostile Environments: Encrypted Computation, Obfuscation, and Other Techniques*

Luis F. G. Sarmenta
lfgs@cag.lcs.mit.edu

Abstract

This paper presents and discusses techniques for protecting executable code from malicious users trying to reverse engineer or modify the code for their own purposes. As platform-independent, easily-decompilable, and mobile code platforms such as Java come into increasingly widespread use, such techniques become necessary not only to guard against software piracy and the theft of intellectual property, but also to protect users of mobile agents from attacks by malicious hosts trying to influence the agents' behavior. In this paper, we discuss three main techniques – *encrypted computation* (also called *mobile cryptography*) [19], *obfuscation* [6], and *time-limited blackbox security* [12]. We present an overview and a critique of each technique, and conclude with some suggestions on possible directions for future research.

1 Introduction

With the growing popularity of Java and other mobile code systems that allow executable code to be automatically downloaded and executed on a user's machine, much attention and research has been focused on protecting users from malicious programs that seek to steal data or do harm to the user's machine. Unfortunately, however, not as much research has been focused on the reverse problem – that of protecting the executable code from attempts by malicious users to reverse engineer or modify the code for their own purposes. This is not so surprising since in the past, reverse engineering from native binary code has generally been a difficult (though not impossible) task. As platform-independent and easily-decompilable binary formats such as Java bytecode become more and more common today, however, one can no longer ignore this problem.

In this paper, we provide a survey of techniques currently being developed to address this problem. We discuss three main techniques – *encrypted computation* (also called *mobile cryptography*) [19], *obfuscation* [6], and *time-limited blackbox security* [12]. We present an overview and a critique of each technique, and conclude with some suggestions on possible directions for future research.

2 Overview

2.1 The Problem

In most general-purpose computer systems today, the code and data of an executable program are exposed to the execution environment, and can be read, disassembled, and modified by a determined hacker. As shown in Fig. 1, this means that once a program owner allows his or her program to run on an untrusted machine, there is usually nothing stopping the owner of that machine from trying to spy on the program's data, manipulate the program's behavior, or damage (sabotage) the program in some way.

The problem, therefore, is to find ways to protect programs from such hostile execution environments.

*Draft v.1.1, July 18, 1999. Submitted as an Area Exam paper, in partial fulfillment of the requirements of the Ph.D. program, Area III, MIT Department of Electrical Engineering and Computer Science.

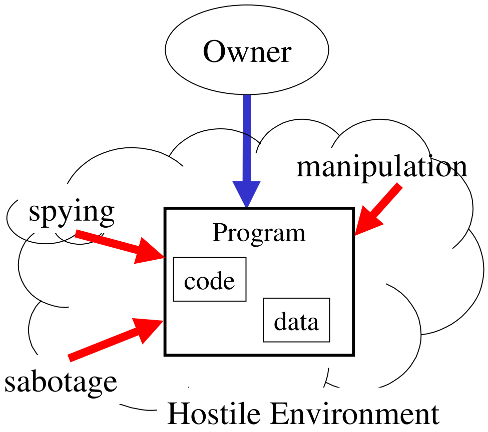

Figure 1: Programs can be attacked in hostile environments.

Ideally, we would like to have *blackbox security*[1] – programs must be protected in such a way that a host executing a program it didn't create:

1. cannot read, or at least cannot understand, the program's code and data, and
2. cannot modify the program's code and data.

That is, all that a host can do would be to feed the blackbox some input, run it, and get some results.

2.2 Applications

An effective blackbox security scheme that would prevent malicious users from reading, understanding, and modifying someone else's programs would have many applications.

Protection of Intellectual Property. The hiding of trade secrets and proprietary algorithms from competitive spying has always been a concern in the software industry. In recent years, the problem has been worsened by the growing popularity of Java [11] and other software platforms that have an easily-decompilable executable format. (For links to some commercial and non-commercial Java decompilers, see [5].) A technique which can somehow prevent others from understanding the decompiled Java code would thus be very useful for hiding such trade secrets. It can also be used as an additional deterrent to *software piracy* by using it to hide the workings of mechanisms for registration keys, trial versions, software watermarks [8], etc.

Protection of Mobile Agents. As more and more people start using autonomous mobile agents, it is becoming vital to find ways to protect these mobile agents from attacks by the hosts on which they run [19, 12]. In an e-commerce application, for example, where an agent may visit different vendors' host machines while shopping for the best offer, there are many ways in which a dishonest vendor may try to manipulate the agent into buying from it [12]. If the malicious vendor can understand and modify the code, for example, he can directly change the behavior of the agent to make it buy from him, regardless of better offers from other vendors. If the vendor cannot change the code directly, but can read and change the data, he can still manipulate the agent by changing the values of critical data, such as the best price seen by the agent so far, in his favor. Worse, the vendor may be able to steal secret information from the agent such as its private keys, digital signing functions, and digital cash, and use these to make purchases and transactions at the agent's owner's expense.

Market-based Parallel Computing. Blackbox security would also be useful in *market-based massively parallel Internet computing* [3], where a parallel computation is done by users offering use of their machines' spare cycles in exchange for compensation. In such cases, we need to prevent dishonest users from trying to get paid without actually doing their work, and from generating erroneous results intended to sabotage the entire parallel computation.

2.3 Possible Solutions

In the following sections, we will examine three different approaches to the problem of protecting programs

[1] The following definition is based on Hohl's definition [12], but extends the idea from agents to programs in general (not just agents), and allows code and data to be read as long as it is not understood.

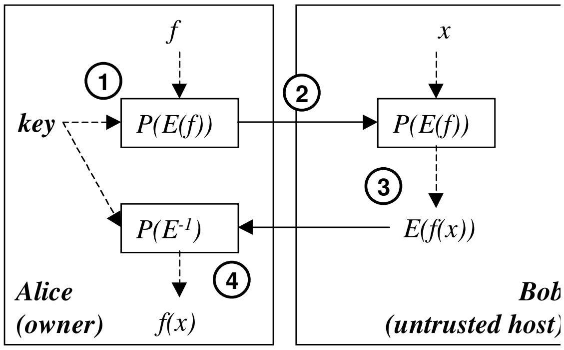

Figure 2: Computing with encrypted functions (CEF).

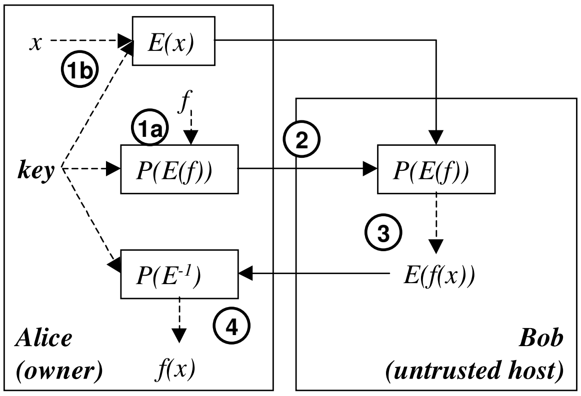

Figure 3: Computing with encrypted data (CED).

from hostile environments: *encrypted computation*, *obfuscation*, and *time-limited blackbox security*. For each of these techniques, we will give an overview, a critique of its strengths and weaknesses, and give some suggestions for future research.

3 Encrypted Computation

Encrypted computation seeks to provide true blackbox security by employing provably secure cryptographic techniques to allow an untrusted host to perform useful computation without understanding what it is computing.

Figure 2 shows one form of encrypted computation. Here, Alice has a secret function f that she wants to run on Bob's machine using some input x provided by Bob. To prevent Bob from understanding or modifying f, Alice encrypts f using her private key and an encryption scheme E to obtain a program $P(E(f))$ for computing $E(f)$ (step 1). She then sends this program to Bob (step 2), who receives the program and runs it on his input x. The program produces the *encrypted* result $E(f(x))$, which Bob sends back to Alice (step 3). Finally, Alice runs a decryption program $P(E^{-1})$ on $E(f(x))$ to obtain the desired value $f(x)$ (step 4). In this way, Bob is able to do useful computation without learning either f or $f(x)$.

This example depicts what is sometimes known as *computing with encrypted functions* (CEF) [18]. Here, f is encrypted, but x is known to Bob. CEF can also be extended to what Sander and Tschudin call *function hiding* [20], where the value of $f(x)$ is returned to Bob after step 4 in Fig. 2. Function hiding can be used to protect intellectual property from theft and piracy. Suppose f is some proprietary algorithm Alice has developed for solving a problem of interest to many people. With function hiding, Alice can sell $(P(E(f)))$ and let users run it on their own machines using their own input. The program would produce encrypted results, which users can then ask Alice to decrypt. In this way, Alice can protect her intellectual property (f), and at the same time charge users for its use through a sort of "pay-per-use" scheme.

Another form of encrypted computation is *computing with encrypted data* (CED), shown in Fig. 3. Here, Alice provides the input data x as well, but encrypts it so that Bob cannot learn its value. (She may or may not allow Bob to understand f.) CED is useful in applications such as market-based parallel computing, where Alice would like to use Bob's computational power without allowing Bob to learn what Alice is doing. Note that CED is can be seen as a special case of CEF if the function f can be redefined to include the data x.

3.1 Basic Examples

As a simple example of encrypted computation, consider the case where $f(x)$ is the function $x + c$. In this case, Alice can encrypt f by simply adding some random number r to c to get the encrypted function $E(f(x)) = x + c'$, where $c' = c + r$. She can then decrypt the output by simply subtracting r from it. Since r can be any number, Bob cannot deduce c from c' and therefore cannot learn the original f. Neither can he manipulate the computation to make it produce a specific output value since he does not know what number Alice would subtract from his output. In this way, Alice is able to protect f from spying and manipulation. Note however, that this scheme is not secure against sabotage. That is, there is nothing preventing Bob from returning a random number to Alice instead of actually performing the computation. Also, this scheme is not efficient since it costs Alice more computation to encrypt and decrpyt the data than to compute the function itself. Thus, it would be cheaper for Alice to just ask Bob for x and compute the function herself.

A similar, but better example is outlined in [21], and shows CED applied to the discrete logarithm problem. This algorithm is much more efficient, since the discrete logarithm computation (which will be solved on Bob's machine) is a hard problem, and is thus likely to dwarf Alice's encryption and decryption overhead. Moreover, Alice can easily check the answer using exponentiation (which is much easier than solving for the discrete logarithm), so the algorithm is safe from sabotage. (Note, that this example is slightly different from the first example since it is doing CED, not CEF.)

3.2 Sander's and Tschudin's work

Recently, Sander and Tschudin, have proposed the application of encrypted computation techniques to the problem of protecting mobile code from malicious hosts [18, 19], as well as protecting intellectual property and secret functions in general [20]. So far, they have developed techniques for the encrypted computation of polynomials. In this section, we give an overview and critique of their techniques.

3.2.1 Polynomial Evaluation using Homomorphic Encryption Schemes

Sander and Tschudin show how encrypted polynomial evaluation can be done using *homomorphic encryption schemes.*

An *algebraically homomorphic* encryption scheme E is one where there exists easily-computable operations PLUS and MULT which take encrypted inputs $E(x)$ and $E(y)$ and can produce encrypted outputs $E(x + y)$ and $E(xy)$ respectively without having to decrypt and learn x and y. With such a scheme, encrypted polynomial evaluation becomes easy – Alice need only give Bob a program where all numerical values in the polynomial (i.e., the coefficients and the input data) are encrypted, and where addition is replaced by PLUS and multiplication is replaced by MULT. Unfortunately, however, no one has yet discovered an algebraically homomorphic encryption scheme.

Sander and Tschudin note, though, that if Bob provides the input x, then Bob does not need a MULT operation, but only a MIXEDMULT operation, which can compute $E(xy)$ using $E(y)$ (from Alice), and x (from Bob), without revealing y or xy to Bob. Since MIXEDMULT in turn can easily be implemented using $O(\log^2 N)$ PLUS operations (by a process similar to longhand binary multiplication), this means that all that is needed to do encrypted polynomial evaluation (on integers, modulo some integer N whose factors are small primes), is an *additively homomorphic* encryption scheme – a scheme which has a PLUS, but not necessarily a MULT operation. Such schemes, unlike algebraically homomorphic schemes, are known to exist [2, 16], and thus make encrypted computation of polynomials possible.

3.2.2 Composition-based approach

Sander and Tschudin also propose another way to perform encrypted computation using *function composition*: suppose f is a rational function (the quotient of two polynomials), and suppose that Alice can generate a random rational function s which is easily invertible. Then, Alice can give Bob the function $B = s \circ f$. When Bob returns $y = B(x)$, Alice can then retrieve $f(x)$ by computing $s^{-1}(y)$. The trick here, of course, is to be able to generate easily-invertible rational functions s at random. Sander and Tschudin cite some known techniques but note that these are not completely secure, and more research needs to be done in this area.

One application of functional composition is in the implementation of *undetachable signatures*. Undetachable signatures address the following problem: suppose Alice has a function f, which she wants Bob to perform on some input x from Bob, and a signing function s, which she wants to run on Bob's machine to sign the output $f(x)$ for her. However, Alice wants to keep s secret so that Bob cannot use it to sign other values (not produced by f) – and thus impersonate her. Using function composition as described above, Alice can give Bob $s \circ f$ instead of s and ask Bob to run both f and $s \circ f$ on his input x to produce $f(x)$ and its signature. Since the decomposition of multivariate rational functions is a hard problem in general, this would prevent Bob from discovering s and generating valid signatures for arbitrary results.

3.2.3 Using Randomization and Redundancy

The encrypted computation techniques discussed so far are susceptible to some attacks. For example, when used in function hiding, the homomorphic encryption techniques are susceptible to *coefficient attacks* [20] that would allow Bob to decrypt the coefficients of f by simply sending Alice the encrypted forms of the coefficients, as if they were the result $f(x)$. The composition scheme for undetachable signatures is prone to many attacks such as *left decomposition* (Bob can determine s given both f and $s \circ f$), *interpolation*, and *inversion* [19]. Sander and Tschudin propose solutions to these problems where the basic idea is to use *tuples* in the functions, where some elements of the tuples are randomly generated by Alice and kept secret from Bob. This randomized information prevents Bob from being able to manipulate the computation to his advantage. It also prevents sabotage, since if Bob randomly changes part of the computation, Alice would be able to detect it, since the "hidden checksum" or signature will not match the output. The cost, of course, is the redundancy and inefficiency introduced by using tuples of random data.

3.3 Critique

Sander and Tschudin's work is very interesting because it gives us hope of achieving true blackbox security *with theoretically provable strength*. That is, unlike the obfuscation techniques to be discussed below, the difficulty of breaking the security of encrypted computation techniques can be predicted formally based on information-theoretic limits or the well-studied computational difficulty of certain mathematical problems. Unfortunately, however, their work in its current state does not yet seem to be useful in real applications for several reasons.

Limited Functionality. First of all, their approach is currently limited to the evaluation of polynomials and rational functions. In fact, it is not clear that they can really handle rational functions either since they note that they still need to come up with a good way to find easily-invertible rational functions. They note that others [1] have developed some theoretical methods for the encrypted evaluation of boolean circuits in general, but that these are very inefficient and require a lot of rounds of communication.

Poor Efficiency. Secondly, their techniques still seem to be too inefficent for practical use. To be practical, the encryption and decryption overhead on Alice's side should not be greater than the cost of computing the non-encrypted function completely on Alice's side. Otherwise, it would be more efficient to just use the *perfectly secure* solution of simply having Bob send Alice his input x and let Alice do all the computation.

Unfortunately, Sander and Tschudin do not provide a formal complexity analysis of their techniques in order to quantify Alice's overhead cost and prove that their techniques are practical. Even if we assume that their techniques can be shown to be practical at least asymptotically, however, there is a question of how big the constant factors are. Since simple integer addition and multiplication can be done very quickly in today's processors, the cost of computing an integer polynomial is quite small, and only becomes significant when dealing with very high-degree polynomials. Thus, it would be interesting to see how large the polynomials have to be before their encrypted computation becomes practical.

The encryption overhead can be amortized away if the same encrypted polynomial is used by Bob on several inputs. This may make encrypted computation more practical. Care must be taken, however, that Bob does not learn much from the values that the encrypted function produces. Sander and Lipton propose an additively homomorphic encryption scheme that claims to be safe in this respect [16]. In the case of function hiding, though, where Alice decrypts $f(x)$ and returns the value to Bob, no matter what encryption scheme is used, Bob can always completely interpolate f by simply asking for $f(x)$ at $k+1$ points, where k is the degree of the polynomial. Thus, in this case, encrypted computation is not practical if the polynomial has to be reused more than k times to make it efficient.

Another source of inefficiency are the techniques needed for hidden checksums, and undetachable signatures. First of all, the redundancy added by using randomized tuples increases the overhead on both Alice's and Bob's sides, as well as the total time it takes to perform the computation. Secondly, the encryption schemes they currently use seem to be very inefficient. The scheme described in the function hiding paper, for example, performs the encryption one bit at a time, and thus seems very inefficient.

Impracticality for Function Hiding and Market-Based Computation. Although Sander's and Tschudin's general idea of using *function hiding* for protecting intellectual property and as a basis for "pay-per-use" computing is very interesting, their example using polynomial evaluation does not seem to be a good one. As noted above, the encryption and decryption overhead seems to be extremely high, such that the only way to make it practical would be to re-use the polynomial many times. However, this would allow Bob to interpolate f, since Bob receives the decrypted $f(x)$ from Alice. Because of these problems, it seems that it would be faster and safer for Alice to just perform the computation completely by herself.

For similar reasons, Sander's and Tschudin's current scheme for encrypted polynomial evaluation does not seem to be practical for market-based computation either. In addition to the overhead on Alice's side which can prevent us from getting any speedup at all from parallelization, there is also the added cost of the PLUS and MIXEDMULT operations compared to simple addition and multiplication, which, while still making parallelization possible, would require a larger number of worker nodes to achieve the same performance. To be fair, however, we should note that Sander and Tschudin did not consider market-based computation as an application in their papers.

Summary of Critique. To summarize, in their current form, Sander's and Tschudin's proposed encrypted computation techniques seem usable, if at all, only for applications requiring the computation of polynomials on the host (Bob's) side where Alice (the server) would like to know the value of some secret rational function $f(x)$ for some input x from Bob, but does not care how long Bob takes to compute it. Applications where Alice would like to achieve speedup, such as in market-based computation, seem unlikely to benefit from these techniques due to the high overheads involved. Applications such as function hiding, where Alice returns decrypted values to Bob need to be carefully analyzed to make sure that: 1) the overhead is less than the cost of having Alice compute everything herself, and 2) Bob cannot learn f or use Alice's decryption "service" for his own purposes. Currently, polynomial evaluation does not seem to be a good example in this respect.

In the future, it would be interesting to see encrypted computation made more efficient and extended to more versatile functions than polynomials and rational functions. Meanwhile, it would also be helpful to do a more formal and comprehensive complexity analysis of Sander's and Tschudin's techniques, as well as come up with real practical applications of their techniques.

4 Obfuscation

Encrypted computation attempts to achieve black-box security by using mathematically-based cryptographic techniques to generate a new encrypted function $E(f)$ from a secret function f, to make it difficult for a malicious user to understand, manipulate, and sabotage the computation of f. Another technique, called *obfuscation*, uses heuristic techniques to modify the *program* for computing f by transforming it into a "messy" form that is very hard to understand, but performs the same function as the original program. While the security properties of obfuscation are not as easy to analyze formally as those of encrypted computation, obfuscation has the advantage of being more generally applicable. That is, while encrypted computation is currently limited to specific problems such as polynomial evaluation, obfuscation automatically applies to any computation that can be expressed as a program.

Obfuscation is an old technique that has been around since programmers first started to worry about protecting their intellectual property from reverse engineering by competitors. In 1992, Cohen studied the application of obfuscation towards protecting operating systems from attacks by hackers or viruses [4]. (At around the same time, these same techniques were actually being used already by virus writers themselves in *polymorphic* or *evolutionary* computer viruses.) More recently, obfuscation has gained a lot of interest with the advent of Java, whose bytecode binary executable format is very easy to decompile back into its original source.

4.1 Obfuscation Transformations

In this light, Collberg, Thomborson, and Low have compiled a taxonomy of obfuscation techniques [6], classifying obfuscation transformations into four general types: *layout*, *control*, *data*, and *preventative* transformations. They further identify four main criteria that can be used to measure the quality of an obfuscation transformation: *potency*, *resiliency*, *stealth*, and *cost*. Potency refers to how much the transformation confuses a human reader, while resiliency refers to how much it confuses an automatic deobfuscator. Stealth [7] refers to how well the obfuscated code blends in with the rest of the program and thus avoids detection and focused attack. Finally, cost can be measured in terms of increased code size and execution time.

4.1.1 Layout Transformations

Layout transformations are transformations that change the way the code *looks* to a human reader. A simple example would be removing the source code formatting information sometimes available from Java byte code. Another example is that of scrambling identifier names to confuse a human reader. The latter is more potent than the former since identifier names carry a lot of pragmatic information that would help human programmers understand the code better. Many popular obfuscators today, such as Crema [23], depend on layout transformations. In general, layout transformations are simple to apply and cost very little in terms of added code size and execution time, but have relatively low potency, since the structure of the code and data are still mostly preserved and can be analyzed and eventually understood despite the change in layout.

4.1.2 Control Transformations

Control transformations aim to obscure the control-flow of the original program. Examples of control transformations include the following:

Computation Tranformations. These transformations change the *apparent* computation being performed by the program. Examples include:

- inserting dead or irrelevant code,
- using more complicated but equivalent loop conditions,
- using language-breaking transformations that take advantage of constructs available in the binary code but not at the source level (e.g., `goto`'s in Java),

- removing library calls and programming idioms,
- implementing a virtual machine with its own instruction set and semantics,
- adding redundant operands, and
- converting a sequential program into a multi-threaded one by using automatic parallelization techniques or by creating dummy processes that do not perform anything relevant.

To these we may add some techniques mentioned by Cohen, such as using different but equivalent machine instructions, and using equivalent instruction sequences [4].

Aggregation Tranformations. These tranformations make code harder to understand by breaking the abstraction barriers represented by procedures and other control structures. Examples include:

- *inlining* subroutine calls and *outlining* sequences of statements (turning them into a subroutine),
- interleaving methods,
- cloning methods, and
- loop transformations.

Ordering Tranformations. These transformations change the order of statements, blocks, or methods, within the code, being careful to respect any data dependencies. These transformations may have low potency, but have high and *one-way* resilience, since in most cases, it is impossible for an automatic deobfuscator to restore the code ordering to its original form. Code ordering transformations have also been proposed as a method for *software watermarking* and *fingerprinting* software [10]. Specifically, the ordering of a program's code can be made into a unique identifier that can be used to track sources of pirated software.

4.1.3 Data Transformations

Data tranformations obscure the data structures used by the application. Examples include the following:

Storage and Encoding Transformations. In many cases, there are "natural" ways to encode and store particular data items in a program. Storage and encoding transformations obscure code by deviating from these and choosing unnatural and encodings for the program's data. Examples of these include:

- changing encoding of primitive data types,
- promoting variables from more specialized forms (e.g. a primitive type) to more generalized ones (e.g., a class),
- splitting variables, and
- converting static to procedural data.

Aggregation Transformations. These tranformations change the way data items are grouped together. Examples include:

- merging scalar variables,
- restructuring arrays, and
- modifying inheritance relations.

Ordering Transformations. These transformations randomize the order in which data is stored in the program. These include

- randomizing the declarations of methods and instance variables within classes,
- randomizing the order of parameters within methods,
- using a mapping function to reorder data within arrays.

4.1.4 Preventative Transformations

Preventative transformations are different from control and data transformations in that they do not aim to confuse a human reader, but to make *automatic* deobfuscation difficult. Examples include the following:

Inherent Preventative Transformations. These transformations introduce problems which are known to be hard-to-solve with known automatic deobfuscation techniques. Some examples include:

- adding aliased formals to prevent *program slicing*,
- adding variable dependencies to prevent program slicing,
- adding bogus data dependencies,
- using opaque predicates with side-effects,
- making opaque predicates using difficult theorems (e.g., `(x*x==(7*y*y-1))` is known to be false for any integers x and y.

Targeted Preventative Transformations. These transformations target known specific weaknesses in existing deobfuscation programs. For example the HoseMocha program [15] inserts extra instructions after `return` statements in the source program. This has no effect on the program execution, but causes the Mocha decompiler [24] to crash.

4.1.5 Opaque Variables and Predicates.

A key tool in implementing these transformations (and especially control transformations) is the idea of *opaque variables* and *opaque predicates*. An opaque variable is one which has some property that is known *a priori* to the obfuscator, but difficult for a deobfuscator to deduce. Similarly, an opaque predicate is one whose outcome is known ahead of time to the obfuscator, but difficult for the deobfuscator to deduce. Figure 4 shows some examples of trivial opaque constructs (from [6]). (Here, we assume `random(a,b)` is a standard library function whose semantics is known to both the obfuscator and the deobfuscator.) These opaque constructs have low resilience since they can easily be broken by an automatic deobfuscator using simple global static analysis.

```
{ int v, a=5, b=6;
  v = a + b; /* v is 11 here. */
  if (b > 5) ... /* True */
  if (random(1,5) < 0) ... /* False */
  if ( ... ) ...
    ... code keeps a and b unchanged ...
  if (b < 7) /* True */
    a++;
  v=(a > 5)?v=b*b:v=b /* v is 36 here */
}
```

Figure 4: Examples of trivial opaque constructs.

Since most deobfuscators are likely to employ static analysis techniques, Collberg et al. propose ways to generate opaque predicates based on problems that these techniques cannot handle well. In particular, they present ways of taking advantage of the intractability of the static analysis of pointer-aliasing problems and multithreaded programs. They note that by manipulating pointers seemingly randomly at many points in the program, for example, we can generate opaque predicates of the form `(p==q)` or `(p==null)`, where `p` and `q` are pointers (object references in Java), whose values would be very hard or impossible to determine statically. And, by employing concurrent threads with carefully controlled, but seemingly random race conditions, we can complicate the process even further.

Opaque variables and predicates can be used in many ways. Opaque predicates, which can be used to direct the control-flow in a program, can be used for many control transformations, such as inserting dead or irrelevant code. Opaque predicates and values can also be used in data transformations to obscure the encoding techniques and mapping functions used in the program.

4.1.6 Building an Obfuscator

Collberg et al. propose to build an obfuscator that will apply all these techniques repeatedly until the desired levels of potency, resilience, stealth, and cost, are achieved. Their proposed obfuscator will not just blindly apply these techniques to the code arbitrarily, but will attempt to analyze the code and determine appropriate transformations to apply to different parts of the code. For example, when obfuscating a part of the code with tight loops, care will be taken to use low-cost transformations, and when obfuscating a computational part without pointers, stealth would be maintained by employing opaque predicates using numbers instead of those based on pointer-aliasing.

4.2 Critique

Collberg et al.'s taxonomy of obfuscation techniques is very impressive. They provide a wide collection of techniques which, it seems, when used together and iteratively as they propose, should make code very, very difficult to understand even with the help of automatic deobfuscators. Furthermore, they identify some concrete formal metrics of program complexity from software engineering research, such as program length, cyclomatic complexity, nesting complexity, data flow complexity, fan-in/out complexity, etc. and propose to measure them and use them in their obfuscation program [6]. This may be a significant advantage in their approach over those of others such as Hohl, who so far only estimate the effectivity of obfuscation techniques using informal analysis.

Measuring Effectivity. The effectivity of their proposed techniques, however, remains to be seen. In general, the main weakness of obfuscation as an approach to blackbox security[2] is the difficulty of measuring how much these transformations really make it harder for a *human* user (fully-armed with deobfuscation tools) to understand the obfuscated code. That is, even if their software is able to produce code that is highly obscured according to some formal metrics, it is still unclear whether or not these would really be effective in protecting the code against a focused attack from determined hackers.

For example, although using pointer aliasing for generating opaque predicates seems like a very promising idea (and one of the significant and unique contributions of their work), the specific techniques (based on graph structures) that they propose for generating these opaque predicates, may not be as effective against human attackers as they are against automatic deobfuscators. A patient and determined human, for example, may be able to notice that some pointers seem to be used in predicates a lot, follow the assignments to these pointers, and learn their opaque values. Also, presumably, the statements in the program that will manipulate the values of these pointers will be generated automatically by an obfuscation program that will follow certain rules to guarantee the desired outcome. If a human can somehow discover these rules, then he could generate a program that uses pattern matching to break all opaque predicates using those rules.

Moreover, often, it is not necessary to discover all the rules and break all the opaque predicates to successfully attack a piece of code. For example, the mechanisms used for enabling or disabling features in trial versions of some software are sometimes based on a few simple `if` statements or on setting a few specific data locations to particular values. Thus, a determined hacker need not deobfuscate the whole program, but only needs to learn enough to determine these crucial statements in the code and focus his attack on them.[3]

Finally, even if the attacker cannot deobfuscate any part of the program at all, it is still possible for him to *sabotage* the program by making random changes in the code or data. Obfuscation alone would not protect against this. (Although it may be used to hide redundant and randomized checksum or signature computations similar in spirit to those used by Sander and Tschudin in encrypted computation).

In any case, it would be useful to look into these questions once Collberg et al. finish Kava, the obfuscation program they are building based on their proposed techniques. Perhaps the best way to test it would be to set up a challenge similar to the RSA decryption challenges [17], where hackers will be invited to crack programs obfuscated by Kava, and cause them to behave a certain way or produce a certain output. This would provide empirical evidence to support their theories, and may even be useful to other researchers in software engineering who are developing formal complexity metrics for software.

[2]Although Collberg et al., like Sander and Tschudin, do not use the term *blackbox security*, which was coined by Hohl, their goals are very similar.

[3]Thanks to Diego Sevilla for pointing this out.

Parameterized Obfuscation. Another, somewhat orthogonal, direction for possible future research is that of developing more concrete schemes for *parameterized obfuscation*. Most current obfuscation programs have fixed strategies for obfuscating a particular piece of code. Such programs would generate the same obfuscated code given the same input program. What would be much more useful would be a program that is capable of generating many different versions of the obfuscated code depending on some parameter such as an integer. Such programs would have many applications. As will be described in the next section, they are necessary for time-limited blackbox security. They can also help track software piracy: a software vendor can sell each new customer a different version generated via parameterized obfuscation using a unique serial number. That way, if the vendor discovers a pirated version, she can trace it to the original purchaser.

Collberg et al. mention this potential application of obfuscation in [6], and note that one can parameterize their obfuscator by choosing the random number seed in the `SelectTransform` algorithm, which selects obfuscation transforms from a pool of appropriate transforms. Different seeds would then lead to different sequences of obfuscation transforms being applied to the code. Since their obfuscator has a large number of transforms to choose from, this technique should be able to produce a large variety of different obfuscated versions of a program – much better than most current obfuscators which can only produce one.

In addition to randomizing or parameterizing the *selection* of transforms, however, it would also be useful to parameterize the workings of *individual* transformations as well. For data transformations, for example, it should be possible to use an integer parameter to generate different encoding maps or order-mapping functions. We can create similar variations within control transformations as well.

As an example, consider one type of control transformation described by Cohen [4], where parts of the original code are replaced by different, but equivalent instruction sequences – e.g., a sequence which performs the computation `x=x+17` can be replaced by a sequence which first performs `x=x+20`, then `x=x-3`. It is easy to see how such a scheme can be parameterized using a single integer parameter p: if the original computation was `x=x+q` for some constant q, then the transformation would generate `x=x+p; x=x-r`, where r is the (precomputed) constant $p-q$. We can parameterize this further by using other integers to determine the type or order of the replacement operations. For example, another parameter can be used to determine whether the subtraction of r is done before or after the addition of p.

Of course, this is just a small example, and it is easy to extend the idea and imagine countless ways to use parameters to generate many differents obfuscated versions of a program. At the highest, most abstract level, such parameterization would be equivalent to Collberg et al.'s idea of simply setting the random number seed in the `SelectTransform` algorithm, since we can view each different version of the transformation (e.g., each p and r in the example above) to be entirely different transformations. However, the difference here is similar to the difference between implementing some function $f(x)$ as a *lookup table* indexed on x, or as a more compact algebraic *formula* based on x – given the same amount of space, the latter can handle a much wider range of inputs, or in the case of an obfuscator, it can produce a much wider variety of obfuscated programs.

It may be that Kava is already being implemented using such parameterized obfuscation. In a paper on software watermarking [8], for example, Collberg et al. present some techniques for encoding integers as graph structures. It is not hard to see that these and similar techniques can also be used as a basis for parameterizable data obfuscation transformations, so it seems safe to assume that they are already using them for this purpose. In any case, though even if they are already using parameterized obfuscation in Kava, a description of how different obfuscation transformations can be parameterized would be a useful supplement to their already-comprehensive taxonomy.

5 Time-limited Blackbox Security

Because they work on computer programs in general, obfuscation techniques are much more widely applicable than encrypted computation techniques, which currently work only for specific mathematical constructs such as polynomials. Thus, at present, obfuscation seems to be the more promising path to achieving blackbox security. Unfortunately, however, as we have seen, obfuscation is not as provably secure as encrypted computation.

Time-limited blackbox security, proposed by Hohl [12], recognizes this shortcoming of obfuscation by restricting the definition of blackbox security defined in Sect. 2.1 as follows: programs must be protected in such a way that a host executing a program it didn't create:

1. cannot read, or at least cannot understand, the program's code and data, and
2. cannot modify the program's code and data
3. *– for a certain known time interval.*
4. *After this time interval, attacks are possible, but these attacks would have no effects.*

Limiting the protection interval in this manner, allows us to use weaker security mechanisms such as obfuscation in implementing time-limited blackbox security.

5.1 Overview

Hohl presents time-limited blackbox security primarily as a way to protect mobile agents from malicious hosts. In this context, time-limited blackbox security can be achieved by first "messing-up" the agent code and data to make it hard to analyze, and then attaching an *expiration date* to indicate its protection interval.

Mess-up Algorithms. Hohl's idea of "mess-up algorithms" is essentially the same as obfuscation. In fact, most of the algorithms he proposes are already mentioned by Collberg et al. (of whom Hohl was not aware). The main idea here is to make the agent's code very hard to understand such that it will take some time before someone can understand it and be able to read it, manipulate it, or sabotage it. This minimum time required to successfully "break" the agent can then be used as the protection interval.

Of course, as in obfuscation, there is still the question of how long it would take a malicious host to break a messed-up agent. Hohl admits that this is an open question, although he points out that we only need to concern ourselves with estimating the time it would take for an *automatic* deobfuscator to break the code, arguing that a human alone cannot possibly do the analysis by himself in time.

Deposited Keys. One mess-up technique proposed by Hohl but not by Collberg et al. is that of *deposited keys*. Here, we encrypt parts of the agent, or remove some critical information needed by the agent and place it on a trusted server. As the agent runs, these parts remain encrypted or missing. Then, when the agent reaches a certain state, it asks the server for the decryption key or the missing parts. In this way, the host cannot analyze the agent before running it and allowing it to come to the required state. (Note: there are some similarities between this technique and that described in [9], which also involves encrypting parts of the program, and decrypting it as the program runs.)

Expiration Dates. After creating an obfuscated version of the agent with the desired protection interval, the agent's owner adds the protection interval to the current time to get an *expiration date*. This expiration date is then included as part of a digitally signed certificate that the agent carries and must present to any other party that it wants to communicate with. Parties who receive such a communication request would first check the certificate to determine if the agent is still valid. If the agent has already expired, communication is refused it, on the assumption that the agent could already have been broken by the malicious host and is thus untrustable. (Note that even if the agent is broken, the certificate itself containing the expiration date still remains valid since it is *cryptographically* signed by the agent's creator and is thus presumably secure for a very long time.) In this way, even if the agent is successfully broken after the expiration date, it cannot cause harm since it cannot communicate with anyone else. (A presumption here is that all receiving parties need to be able to get the correct global time in order to check the expiration date.)

If the agent to be sent out contains some *token* data – i.e., self-contained documents that depend on the identity of the owner and can be traded independently of the issuer, such as digital cash, secret keys, etc. – then the owner must also attach expiration dates on these tokens and digitally sign them. He must do this *before* the agent is obfuscated so that the obfuscation algorithms will also scramble the way the tokens are stored in the agent and ensure that a malicious host cannot get the tokens before the protection interval. Then, even if a malicious host successfully breaks an agent *after* the protection interval and manages to get its tokens, no harm is done, since the tokens would already be expired and unusable.

Finally, note that the expiration date protects the agent even if it migrates to another host. And, if the agent needs to extend its life for some reason, then it can get "recharged" by returning to its owner (or some other trusted host), which can then *reobfuscate* it using different parameters, and then give it a new expiry date.

Limitations and Open Problems. At present, the main problem in implementing time-limited blackbox security is the difficulty of estimating and guaranteeing the protection interval of an obfuscated agent. However, Hohl also recognizes some other limitations and open problems, including the following:

- There is a need for a standard *global* time that can be used to check expiration dates. This requires either some mechanism for distributed clock synchronization, or communication with a trusted timekeeper, which may increase communication overhead and limit scalability.
- The types of tokens that can be protected using expiration dates are limited. For example, tokens which need to be protected past the expiration date (e.g., the owner's long-term private keys) should not be carried by time-limited agents.
- Obfuscation leads to longer transmission and execution time for agents, which may limit the applicability of this technique to applications where speed is not a critical factor.
- Time-limited blackbox security (and blackbox security in general) may still be susceptible to *sabotage*, where the host destroys parts of the agent, and *blackbox test* attacks, where the host tries to deduce the internal workings of the agent by running it many times on many different combinations of inputs. Hohl notes that we may be able to protect against the former by embedding checksum code within the algorithm. If the agent is truly a blackbox, then sabotage is likely to lead to random damage, and can thus be detected by checking the checksums.

5.2 Critique

The idea of time-limited blackbox security is a very promising one. It aims to achieve the same goals sought by Sander and Tschudin in mobile cryptography, but seems to offer a much better chance at actually becoming implementable and usable in the near future.

Better Obfuscation, Deposited Keys, and Interactive Techniques. Hohl's list of mess-up algorithms is somewhat limited, and can benefit a lot from incorporating Collberg et al.'s obfuscation transformations, especially if the latter can be parameterized as suggested earlier.

At the same time, it would also be useful to further study and develop the idea of deposited keys and similar techniques. This idea is different from the other techniques discussed by Sander and Collberg in that it is *interactive*. This is both an advantage and a disadvantage. The disadvantage is that it requires communication with a trusted server, and can thus increase execution time and limit scalability.

The advantage, however, is that one can rely on more *information-theoretic*, and not just complexity-based security against attacks. That is, if parts of the code are actually missing and cannot be predicted, then it is not possible to analyze the code *regardless of how much time an attacker may have.*

Sander and Tschudin seem to think that the cost of extra communication outweighs the advantages of interactive techniques, and thus consider only non-interactive techniques for their mobile cryptography. There may be applications, however, where the extra communication may be acceptable. For example, if an agent needs to communicate with the server at certain points anyway, then we can piggyback the deposited keys with the normal communication packets at these points. In this way, we can use interactive techniques to provide significantly more security at minimal additional cost.

Applications. In his paper, Hohl focuses exclusively on using time-limited blackbox security for protecting mobile agents against malicious hosts. It would be interesting to see if his ideas can also be applied to other applications as well, such as the protection of programs and intellectual property from reverse engineering and piracy, and market-based parallel computation.

As is, time-limited blackbox security does not seem directly applicable to the problem of protecting software from piracy and intellectual property theft. Unlike mobile agents, which are typically sent out to other hosts for relatively specific and short tasks (e.g., to find the best deal on an item), software has a much longer intended life and thus needs a longer protection interval. Unfortunately, the task of guaranteeing a protection interval by using obfuscation becomes much harder for longer intervals. This is because as the intervals get longer, the human advantage over automatic deobfuscators becomes a more significant factor. If a program's time limit is in the order of seconds or minutes, for example, (a reasonable time for a mobile agent to achieve its task), then it is safe to assume, as Hohl does, that we only need to worry about automatic deobfuscators since it is simply impossible for a human to process the information in such a short time. It takes time, for example, for a human just to *read* the text of the program, let alone understand it and modify it. If the program's time limit needs to be extended to the order of months, or even just days, however, then considering the human ability to deobfuscate code becomes necessary. This is a much harder problem. While complexity analysis may be used to predict the time it would take for certain deobfuscation algorithms to unravel a piece of obfuscated code, it is much harder to analyze the human thinking process in a similar way.

It is also important to note that time-limited blackbox security applied to a program does not mean that the program becomes unusable or self-destructs after the protection interval. On the contrary, it means that we assume the user can do *anything* it wants with the program after the protection interval. This presents another problem if the program we are trying to protect is a commercial software package and not just an agent. With an agent, we are able to guarantee that no harm is done after the protection interval by simply preventing other parties from communicating with the agent. Thus, even if the malicious host is able to understand all the algorithms contained in the agent, it cannot use them. In contrast, with commercial software, if a malicious user is able to reverse engineer a program after the protection interval, then harm has already been done. The user can then, for example, rewrite the program, remove its security mechanisms, and then sell copies of the modified program to others. Since many commercial programs do not depend on communication with other parties to operate (e.g., word processors), there is no way to prevent the broken programs from being used in the same way that we prevented expired agents from being used.[4]

Fortunately, while time-limited blackbox security cannot be used for protecting intellectual property, it can be useful for market-based parallel computing. The idea here is to give the worker hosts obfuscated and time-limited code for performing part of the parallel computation. The worker runs the code until the

[4] One way to make time-limited blackbox security work here may be to *require* even ordinarily non-communicative programs to communicate with a server to operate. This leads to an idea similar to Sander and Tschudin's function hiding.

time limit, at which point, the server gives it a new reobfuscated version of the same code (or the code for a different part of the parallel computation). Here, as with mobile agents, the code given to the hosts is specific to the current problem being solved, and has a relatively short lifetime. Thus, it should be easier to predict and guarantee protection intervals for it. Furthermore, as with agents, we can prevent malicious hosts from doing harm by simply ignoring any communication requests or any results produced by expired code. Of course, the server should also take care to change the data encoding, or to the change the computation itself, when giving new code to the worker after the previous time limit expires. In this way, even if the malicious worker understands the previous version of the code, he cannot use that knowledge to affect the results of the new version.

It is important to note, however, that the cost of obfuscation is a more significant factor when applying time-limited blackbox security to market-based parallel computation. The need to periodically give new code to the workers incurs a certain amount of communication overhead. In the case of Java code, it can also mean a reduction in performance since frequently changing the code prevents us from taking full advantage of just-in-time (JIT) compilation techniques, which take a certain warm-up time, but speed up the code execution considerably. In applications where the correctness and security of the computation is a major concern, however, the cost may be worth it. It may be better, for example, to have the computation take 10 times longer but be guaranteed correct, for example, than to risk having a small error due to one sabotaged worker invalidate and waste the work of 1000 other workers.

6 Suggestions for Future Research

The problem of protecting programs from attacks by malicious users is a relatively new research area, and there is much space left for further exploration. Some ideas for further research have already been mentioned, including the following:

- *Parameterized Obfuscation.* It would be useful to describe concrete ways not only to *select* between different obfuscation transformations as Collberg et al. suggest, but also to parameterize the *individual* transformations themselves. This (in addition to Collberg et al.'s parameterized selection) would lead to being able to generate more variations on a piece of code, and would be useful for implementing time-limited obfuscation.

- *Holding a cracking contest to measure effectivity.* The main disadvantage of obfuscation is that unlike mathematically-analyzable techniques such as encrypted computation, it is hard to predict the effectivity of obfuscation transformations. One way to *empirically* measure such effectivity may be to hold a publicized contest inviting hackers to crack a certain obfuscated program and make it behave in a certain way. This would be similar to the successful decryption contests held by RSA [17].

- *Interactive techniques.* Sander and Collberg consider mostly non-interactive techniques, in order to avoid the extra cost incurred by requiring the program to communicate with the server. In cases where such communication is already required anyway, then it might be profitable to apply interactive techniques, such as Hohl's deposited keys idea, to achieve much better security.

In addition to these, there are at least two other general ideas which deserve further study: using redundancy and randomization, and combining encrypted computation and obfuscation.

6.1 Redundancy and Randomization

A potentially very useful technique is that of using redundancy and randomization as a way of further enhancing security. For example, Sander and Tschudin use such techniques when addressing the problems with undetachable signatures and function hiding [19, 20]. Their solutions work by giving the host a new function where each coefficient is replaced by a *tuple* containing elements not only from the original function, but also from other randomly-chosen functions whose answers are either known, or can easily be evaluated. This makes it much harder for an attacker to understand the structure of the original function or to generate a manipulated answer. Unless the attacker can deduce *all* the other functions, *and* identify which one is the original one, any changes that the attacker makes to the function is likely to cause an inconsistency in the answer that the owner can detect. Similar techniques may be used in time-limited blackbox agents to implement the embedded checksums suggested by Hohl as a protection against sabotage attacks.

Further research in this area may also benefit from the results of research in *digital watermarking*, and specifically, of the watermarking of *active intellectual property* such as software and hardware designs.[5] The idea behind such watermarking is to embed some property in the software or hardware design that is checkable by the owner using certain key information, but unnoticeable to people without the required key information. The watermark should also be resilient to attempts to remove it from the watermarked object. Collberg et al. classify some common watermarking techniques for software and suggest some of their own in [8]. A group at UCLA is also currently studying the watermarking of hardware designs and solutions to graph theory problems [13, 14, 22].

Most watermarking techniques are designed primarily as protection against intellectual property theft, and thus probably cannot be applied directly in their present form to the problem of hiding checksum computations in blackbox code. However, some ideas from these techniques may be helpful. One technique presented in [14], for example, protects an FPGA design by randomly selecting a number of normally invisible (internal) nodes to be made visible in addition to many more normally visible nodes. This allows the owner to check for design theft, since the chance that another honest designer would come up with a design that includes not only the normally visible nodes, but also the same set of normally invisible nodes is very small. While this exact implementation may not be directly applicable to blackbox security, similar techniques involving the addition of a randomized set of unnecessary information may be useful.

Another example of a watermarking technique that may be useful is that of *constraint-based watermarking* [13]. Here, additional constraints are added to the original problem based on a certain private key. A solution is then generated to fulfill these and the original constraints. Because of the additional constraints, the resulting solution space would be restricted. Checking for the watermark can then be done by checking if a certain solution satisfies the additional constraints. If the original solution space is large, and the constraints are chosen randomly,[6] then the chances of an honest designer coming up with his own solution to the original problem which also happens to satisfy the extra constraints will be small.

A similar technique may be used for protecting mobile agents and market-based parallel computing systems against sabotage by malicious hosts. The idea is to ask the hosts to solve a *harder* (more constrained) problem whose answers will have properties that the server can easily check. Thus, if a host's answer does not contain the desired property, we can assume that it did not run the code it was given correctly. Sander and Tschudin's idea of using tuples may be considered one example of this idea.

6.2 Combining Encrypted Computation and Obfuscation

Finally, it would also be interesting to find ways to combine the ideas of encrypted computation and obfuscation to get better security.

In looking for such solutions, it would probably be helpful to assume that the server expects to receive *encrypted* data from the host, and the host does not need to understand the results it produces. This is because allowing the host to understand the meaning of even just the results makes the scheme susceptible to many different learning attacks such as

[5] as opposed to the watermarking of *artifacts* such as digital images.

[6] but carefully, to avoid constraining the problem too much that it becomes unsolvable.

interpolation and the coefficient attack, noted ear in Sect. 3.2. Fortunately, this assumption is not restricting. Although it does not allow us to prot commercial software from piracy (since commer software in general are designed to produce un crypted data for the host's use), it can still be usec mobile agent applications, and in market-based p allel computation, where the host does not need understand the results generated by the code.

Given this assumption, some possible approac include: obfuscating encrypted functions, using cryption as a part of the obfuscation process, using obfuscation transforms as encryption schem

6.2.1 Obfuscating Encrypted Functions

One straightforward way to combine encrypted computation and obfuscation is simply to first use encrypted computation to encrypt the function to be computed (assuming it is a polynomial or rational function), and then obfuscate the program afterwards. By obscuring the workings of the PLUS operator, and hiding the coefficients, this would add an extra layer of security and make it harder (though not impossible) for a malicious host to take advantage of attacks such as coefficient attacks, which are based on knowledge of the encrypted coefficient values and the structure of the function to be evaluated.

6.2.2 Encryption as Part of the Obfuscation Process.

It may also be possible somehow to use encryption techniques to determine the reordering and regrouping of the program's code and data. In [8], for example, Collberg et al. talk about watermarking software by taking two secret primes, encoding their product as graph structure, and hiding the structure in the program heap among other structures. It is not yet clear, however, if and how this idea can be applied to achieving blackbox security.

6.2.3 Obfuscation as Encryption

Finally, a most intriguing and promising possibility is that of using obfuscation itself as a form of encryption.

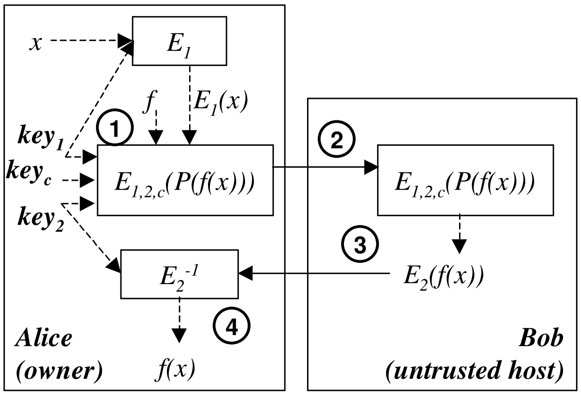

Figure 5: CED using obfuscation as an encryption scheme.

Implementation. Suppose, for example, that we consider data obfuscation transformations, such as changing data encoding, splitting and merging variables, and reordering and restructuring arrays and other data structures, as encryption schemes. Then, we can do computing with encrypted data (CED) as shown in Fig. 5. Here, Alice chooses two secret "keys" or parameters and uses them to select (or generate) two data obfuscation schemes, E_1 and E_2. E_1 is used for encoding the input data, while E_2 is used for encoding the output. Using these keys, together with a code obfuscation parameter, key_c (for added variety), Alice then generates a program (step 1), $E_{1,2,c}(P(f(x)))$, which encodes and stores x according to E_1 and works on it to produce a result encoded according to E_2 without revealing the original value of x. Alice gives this program to Bob (step 2), who runs it on his own machine to produce the encoded answer $E_2(f(x))$ (step 3). Bob then returns the answer to Alice, who runs the decoding program $P({E_2}^{-1})$ on Bob's result to get the original value of $f(x)$ (step 4). As in the CED example described in Sect. 3.2, Bob only sees encrypted forms of the data and the code, and thus never learns x or $f(x)$.

While this scheme is very similar to those described in Sect. 3.2, there are a few important differences.

First, unlike the techniques in Sect. 3.2, this scheme is not limited to using algebraic functions and algorithms to perform the encrypted (homomorphic) operations. Thus, while no one has yet found a tradi tional encryption scheme where one can perform *bot* the PLUS and MULT operations homorphically – le alone perform generalized computation – doing so i relatively straightforward with obfuscation. Since th program $P(E_{1,2,c}(f(x))$ can be *any* program that ca be expressed as a sequence of machine instructions we know that we can implement any operator tha we need as long as it is computable.

Second, note that in Sander's and Tschudin' schemes, the homomorphic operators PLUS an MIXEDMULT where *fixed* and *well-known*, even t Bob. It is only the data's encrypted form tha changes, based on the private key. In contrast, here not only the data, but also the code given to Bob change depending on the private keys.

Finally, since $P(E_{1,2,c}(f(x))$ is not limited to be ing mathematically describable and fixed, its operations do not have to be homomorphic. That is, as shown in Fig. 5, the encoding scheme for the result, E_2, does not have to be the same as the encoding scheme for the input E_1. In Sander's and Tschudin's scheme, the PLUS operator had to be homomorphic because it was used many times on Bob's side in the process of calculating the polynomial. In fact, finding a way to do encrypted polynomial evaluation using just one homomorphic operator (PLUS) was one of Sander and Tschudin's major achievements, since homomorphic operators, and schemes which support several of them, are generally hard to find. If, as described above, however, we have complete freedom in writing the encrypted program $P(E_{1,2,c}(f(x))$, then limiting ourselves to homomorphic operators is not necessary, and we may be able to get more flexibility and more security by allowing the output to be encoded differently from the input.

Example. Figure 6 shows an example. Here, Alice wants to hire Bob to compute the following computation for her without allowing Bob to understand the computation,

```
a = b + c;
b = b + 1;
c = a * b;
```

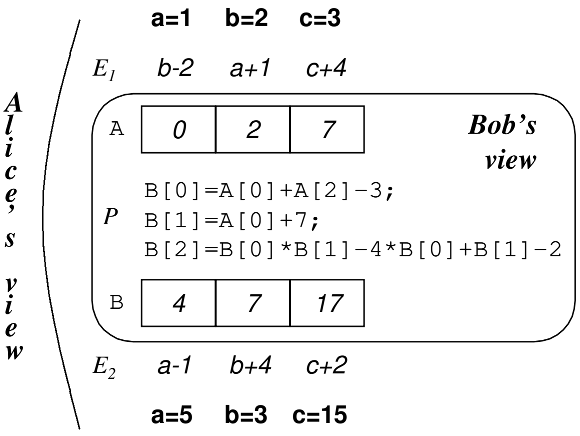

Figure 6: Computing `a=b+c;b=b+1;c=a*b;` using obfuscated data.

As shown, she first chooses two different encoding schemes – E_1 for the input, and E_2 for the output – which scramble the data by adding random numbers to them and placing them in an array in random order. Given these two schemes, she then generates a program, P, to perform the desired computation using the new encodings, and gives this program to Bob together with the encoded input array A. (Note that Alice can make it even harder for Bob to understand the computation by employing more complex data encoding schemes as well as applying code obfuscation transformations on P.) Bob then runs the program to produce the output array B, which he returns to Alice. Finally, Alice computes the inverse of E_2 on B to get the desired values of a, b, and c.

By thus allowing Bob to see only encoded versions of the data and the code, Alice is able to make it hard, or even impossible, for Bob to understand the computation. Moreover, unlike in traditional encrypted computation, Alice can have Bob perform *arbitrary* secret computations for her, since all she needs to do is to rewrite the original computation to work with the scrambled data. This example, for instance, shows Alice having Bob perform a secret computation using addition and multiplication operations without the need for an algebraically homomorphic encryption scheme.

Of course, since this scheme is based on obfuscation techniques, the problem of predicting its security remains a problem. There is one significant advantage here, however. In most traditional applications of obfuscation, such as in protecting commercial software programs, the obfuscated program has to receive *unencoded* data from the user, process it, and then produce *unencoded* output. This means that even if most of the internal code and data are obfuscated, there is still at least some code needed for converting between the unencoded and encoded forms of the data. If Bob can identify and understand this part of the code, he can understand the data encoding, and gain a much better chance of understanding the code itself. In constrast, the scheme in Fig. 6 never allows Bob to see any data in unencoded form. Thus, even if he can understand how P works, he can never be totally sure that his understanding is correct.

For example, suppose, using a process like the one to be described below, he is able to analyze P and come to the conclusion that the code is performing a computation which looks like:

```
a = b + c;
b = b + 1;
c = a * b;
```

He still cannot be sure of this analysis, since if the array orderings were slightly different, the computation can also mean:

```
c = b + a;
b = b + 1;
a = c * b;
```

(with a and c swapped), which would have a different meaning if a and c are used in other parts of the program. Of course, in reality, Bob's job is even harder, since the constant numbers added to the variables can be random, so he can never be sure whether each statements involves constants, and if so, what their values are.

Caveats. It is important to note that although hiding the data encoding schemes from Bob makes it much harder for Bob to understand the obfuscated program P, Bob may still be able to break the code given some information about the code. For example, if he knows that the pattern $b = b + 1$ is common in the program, and that other similar patterns such as $b = a + 1$ or $b = a + 3$ are very rare, then he may be able to deduce that $A[1]$ contains an encoded version of the original b and $B[1]$ will hold an encoded version of $b + 1$. If he further knows that Alice has the habit of writing code such that statements with additions and multiplications involving two variables never involve constants, then he may be able to deduce the structure of the first and third statements, and then deduce the constants in the encoding. Finally, by looking at other parts of the code, he may be able to deduce that Alice is more likely to compute `c=a*b` than `a=c*b` and thus identify the proper array ordering.

This problem is similar to that of old cryptographic techniques such as the substitution cipher, which can be broken by noticing the frequencies of different encoded characters and matching them with known frequencies of letters in the original language. Modern-day cryptographers have gotten around this problem by developing techniques designed to produce random-looking ciphertexts without the statistical patterns of the original language. Applying the same idea to obfuscation, this means that we should choose data and code obfuscation transformation schemes that would hide statistical patterns in the original code. Ideally, an obfuscated program P, should be the *possible* result of *many* different original code sequences, each *with equal probability.* In this way, even if Bob is able to generate all possible deobfuscations of P, he still will not know which one is the correct one, since they are equally likely.

A further way to mask statistical patterns is to interleave different and independent computations. For example, to prevent Bob from deducing the structure of the code above, Alice may choose to interleave it with the code for solving another problem where b is not always incremented by 1, multiplications and additions can involve constants as well as variables, and a is more likely than c to contain a product.

In general, suppose Alice has two different and independent computational problems C_1 and C_2, which taken several steps to compute. She can interleave the computations either:

- in space – at each step, give Bob a program with the code for a step in C_1 and a step in C_2.

- in time – at each step, give Bob a program with the code for a step in either C_1 or C_2, but without letting Bob know which one.

- or both – at each step, give Bob a program which may perform zero or more steps from either C_1 or C_2.

Interleaving code in either space or time helps hide statistical patterns in C_1 and C_2 if these have different patterns. Interleaving computations in time makes analysis even harder by obscuring the data flow in the code. Suppose for example that a program Bob receives for a step writes to a variable a, and the program for the next step reads and performs computation on a. Bob cannot conclude anything from this since the second program may actually belong to a totally independent computation where a has a different meaning and value.

Summary. Whether these and other techniques would be enough to make obfuscation feasible as a way of doing encrypted computation is still an open question, but it looks very promising at the moment and thus deserves further investigation. At the very least, hiding both the input and output data encoding schemes from the host (Bob), will almost certainly make deobfuscation significantly more difficult, and can thus be used as a way of increasing the protection intervals needed in time-limited blackbox security.

7 Conclusions

In this paper, we discussed different ways of protecting programs such as commercial software and mobile agents from attempts by malicious users to understand, modify, and manipulate their behavior. Using mathematical principles, encrypted computation promises the potential of finding provably secure ways to hide the details of a function from the host executing it, but currently works only for a few specialized functions such as polynomial evaluation, and seems to be too inefficient to be practical in real applications. Obfuscation offers more versatility and efficiency, but its security properties are harder to analyze due to the heuristic nature of both the obfuscation and deobfuscation processes. Time-limited blackbox security recognizes this limitation in obfuscation by preventing mobile code from communicating with other parties after a certain protection interval based on an estimate of the time it would take to deobfuscate the code.

All these techniques are relatively new, and provide a lot of opportunity for further research. Some ideas worth looking into include parameterized obfuscation, interactive techniques, using redundancy and randomization techniques such as checksums and watermarks, and combining encrypted computation and obfuscation. In the end, a successful solution to this problem will probably be achieved by combining all these techniques – that is, by using parameterized obfuscation as an encryption scheme (with randomized parameters as the private keys) to prevent the malicious host from understanding and manipulating the code, and by using checksums and watermarks to detect attempts to change the code.

Acknowledgments

Many thanks to Diego Sevilla of the Universidad de Murcia in Spain for pointing me to Collberg's obfuscation pages, and to Christian Collberg and Clark Thomborson for pointing me to work by others in this area.

References

[1] Abadi, M. and Feigenbaum, J.: Secure circuit evaluation. *Journal of Cryptology*, 2(1):1-12, 1990.

[2] Benaloh, J.: Dense probabilistic encryption. *Proc. of the Workshop on Selected Areas of Cryptography*, 1994, pp. 120-128.

[3] Cappello, P., Christiansen, B., Neary, M.O., and Schauser, K.E.: Market-Based Massively Parallel

Internet Computing. *Proc. Third Working Conf. on Massively Parallel Programming Models*, pages 118-129, London, UK, Nov. 1997.

[4] Cohen, F.B.: Operating System Protection Through Program Evolution. 1992. `http://all.net/books/IP/evolve.html`

[5] Collberg, C.: Christian Collberg's Software Watermarking and Obfuscation Page. `http://www.cs.arizona.edu/~collberg/Research/Obfuscation/`

[6] Collberg, C., Thomborson, C., and Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report 161, Department of Computer Science, The University of Auckland. New Zealand, July 1997.

[7] Collberg, C., Thomborson, C., and Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *Principles of Programming Languages 1999 (POPL'99)*. San Diego, CA, Jan. 1998.

[8] Collberg, C., and Thomborson, C.: Software Watermarking: Models and Dynamic Embeddings. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*. San Antonio, Texas, Jan. 1999.

[9] Cooper, T.E., Philips, H.W., and Pryor, R.F.: Method and apparatus for enabling trial period use of software products: Method and apparatus for utilizing a decryption block. US Patent 5,598,470, Jan. 1997. Assignee: IBM Corporation.

[10] Davidson, R.L., and Myhrvold, N.: Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Sept. 1996. Assignee: Microsoft Corporation.

[11] Gosling, J., Joy, B., Steele, G.: *The Java Language Specification*. Addison-Wesley, 1996. `http://java.sun.com/`

[12] Hohl, F.: Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts, in Vigna, G. (ed): Mobile Agents and Security, Springer-Verlag, Germany, 1998.

[13] Kahng, A.B., Lach, J., Mangione-Smith, W.H., Mantik, S., Markov, I.L., Potkonjak, M., Tucker, P., Wang, H., and Wolfe, G.: Watermarking Techniques for Intellectual Property Protection. *35th Design Automation Conference Proceedings*, pp. 776-781, 1998.

[14] Kirovski, D., Huang, Y-Y., Potkonjak, M., and Cong, J.: Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions. *IEEE/ACM International Conference on Computer Aided Design*, pp. 194-198, San Jose, California, November 1998.

[15] LaDue, M.D.: HoseMocha. `http://www.xynyz.demon.nl/java/HoseMocha.java`, Jan. 1997.

[16] Lipton, R. and Sander, T.: An additively homomorphic encryption scheme or how to introduce a partial trapdoor in the discrete log. Submitted for publication, Nov. 1997.

[17] RSA Data Security: RSA Laboratories Challenges. `http://www.rsa.com/rsalabs/html/challenges.html`

[18] Sander, T. and Tschudin, C.F.: Protecting Mobile Agents Against Malicious Hosts; In *Mobile Agents and Security*. Springer LNCS 1419. `http://www.icsi.berkeley.edu/~sander/publications/MA-protect.ps`

[19] Sander, T., and Tsuchdin, C.F.: Towards Mobile Cryptography. Proc. of Security and Privacy '98. Oakland, California, May 1998. `http://www.icsi.berkeley.edu/~sander/publications/satschu.ps`

[20] Sander, T., and Tschudin, C.F.: On Software Protection via Function Hiding. In *Proc. of the 2nd Workshop on Information Hiding*, Springer LNCS. `http://www.icsi.berkeley.edu/~sander/publications/hiding.ps`

[21] Schneier, B.: Applied Cryptography. 2nd ed. John Wiley & Sons. (1996)

[22] UCLA IPP Group Publications Page. `http://www.cs.ucla.edu/~gangqu/ipp/`

[23] Van Vliet, H.P.: Crema – The Java obfuscator. `http://web.inter.nl.net/users/H.P.van.Vliet/crema.html`

[24] Van Vliet, H.P.: Mocha – The Java decompiler. `http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html`. Also at: `http://www.brouhaha.com/~eric/computers/mocha.html`