Can We Obfuscate Programs?

Boaz Barak

The paper [1] claims some impossibility results for the task of obfuscating computer programs, and it has generated some discussion, and some confusion (e.g., see [2]). In this essay I try, as one of the authors of that paper, to explain (my opinion of) what these results mean and what they do not mean.

What is a Program Obfuscator?

A program obfuscator is a sort of compiler that takes a program as input and produces another program as output. We call the output program the *obfuscated program*. (Note that as in a standard compiler, the input program and the obfuscated program do not have to be in the same programming language. For example, the input program can be in C while the output program can be in machine language. However, it turns out that this doesn't make much of a difference.) The three conditions we want from an obfuscator are the following:

- **Functionality:** The obfuscated program should have the same functionality (that is, input/output behavior) as the input program. For simplicity, we concentrate here on programs that compute a single non-interactive *function*. For such programs, the functionality requirement is that the input program and the obfuscated program should compute the same program.

- **Efficiency:** The obfuscated program shouldn't be much less efficient than the input program. We are willing to allow some overhead in the obfuscated program, but it shouldn't be the case that it runs in time, say, exponentially slower than the input program.

- **Obfuscation:** This is the most important condition and the hardest one to define. At this point, let us say that the obfuscated program should indeed be somewhat ``obfuscated''. This means that even if the code of the original program was very easy to understand, say, possessing meaningful comments, function and variable names, and a modular design, then still the code of the obfuscated program should be hard to understand. Presumably, any hacker looking at the code of the obfuscated program should gain nothing more than a headache.

I don't want to elaborate too much as to why are people interested in obfuscators and why would someone think they might exist. I will just say that obfuscators can be very useful for many applications, in particular software protection and digital rights management (DRM), and that I believe that anyone who has ever tried to figure out someone else's code can understand why obfuscators may exist (at least human ones... **:)**). For more discussion, see the paper [1].

Completely breaking an obfuscator.

Without defining what is exactly the obfuscation property, there is certainly one minimum requirement: a hacker should not be able to recover completely the source code of the input program from the obfuscated program. If a hacker manages to recover the source code of a program `P` from the obfuscated program, then we say that the obfuscator *completely failed* on the program `P`. Now there are two immediate observations:

- **It's easy to avoid complete failure for *some* programs:** Consider the "comment stripping" obfuscator. That is, a compiler that the only thing it does is to erase all the comments from the input program. This obfuscator clearly satisfies the functionality and efficiency conditions. In addition, there are *some* programs that if we obfuscate them with the "comment stripping" obfuscator then a hacker will not be able recover the source code from the obfuscated program. For example, this will be the case if the original program contained a secret number in the comment, and this secret number is not referred to in any other place in the code.

- ***Every* obfuscator completely fails on *some* programs:** Consider a program that prints its own source code. (Constructing such programs is a nice exercise often given in undergraduate programming courses.) If an obfuscator satisfies the functionality requirement, then any hacker by just executing the obfuscated program, will obtain a printout of the original source code. This can be generalized as follows: we say that a program is *learnable* if it is possible to recover its original source code by just executing it (on different inputs). Clearly, every obfuscator will fail on learnable programs, however this is not really an "interesting" failure. Also, the functions one wants to obfuscate in practice (such as hash functions or block ciphers) are usually very far from being learnable.

What is the result of [1]?

The main result of [1] is the following: There exists a program (actually a family of programs, but let's ignore that here) such that: on the one hand it is (strongly) non-learnable, but on the other hand *every* obfuscator fails completely when given that program as input.

- **In what way is this result strong?** This result is strong in the sense that the obfuscator fails *completely* on these programs. Its not that the obfuscator manages to hide some partial information, but rather, a hacker can completely recover the original source code from any supposedly obfuscated version. It is also strong in the sense that these programs are strongly non-learnable, and are actually very close variants to the kind of functions people want to obfuscate in practice.

- **In what way is this result weak?** This result is weak in the sense that it only shows that every obfuscator fails completely on *some* (unlearnable) programs. Of course, in some sense, this is the best we can hope for, because even the simple comment stripping obfuscator

doesn't fail completely on *all* the programs. Still, a commercial obfuscator manufacturer can justifiably say that his customers are not interested in obfuscating the family of programs constructed in [1]. They are interested in obfuscating *their* programs, and we haven't shown that this will be insecure.

What does this result mean?

To understand what this result means one needs to understand the two different categories of cryptographic constructions that exist today. On the flip side, understaing this is very beneficial, and is probably much more important than understading these results on obfuscation.

I like to divide all cryptographic constructions, be they encryption, digital signature, hash functions, or obfuscators, into two categories, depending on the level of security they posses. I call these categories **well-defined security** and **fuzzy security**.

Well-defined security. A good example for well defined security is *digital signatures*. Today, the standard definition for secure digital signature is *existential unforgeability*. Roughly speaking, this means that even if a hacker gets access to signatures on many messages of her choice, she will still not be able to forge a signature even for a *single* new message. Now this definition seems quite stronger than what is necessary. Indeed, in most applications where digital signatures are used, it is unlikely that a hacker will be able to obtain signatures to messages of her choice. Also, forging a signature on an arbitrary message, would not really help her break the system, unless this message is *meaningful*. For example, in some applications it will be useless for her to forge a signature on a message that is not in English, while in others it will be useless for her to forge a signature on a message that is not in well-formed XML syntax. Even though this existential unforgeability definition is quite strong, it is considered to be the ``right'' definition of security, because it allows us plug in digital signatures in various applications, and rest assured that the type of messages that are used in the application, be it English, French or XML, will not cause a security problem. Indeed, constructing secure applications is complicated enough without having to worry about digital signatures with subtle weaknesses. Also, there are in fact several known constructions that are widely believed to satisfy this strong definition. It's true that none of them are *unconditionally proven* to satisfy it (yet), but for some of them we *can* show that if there exist a forger then there exist an algorithm for a known hard computational problem (such as the problem of efficiently factoring large integers).

Relationship with proven security. Although these notions are related, well-defined security should not be confused with *proven* or *provable* security. The well-defined adjective refers to the cryptographic *concept*, such as digital signatures, public-key encryptions, or collision-resistant hash functions. The *provable* adjective refers to a particular implementation such as the Cramer-Shoup or RSA cryptosystem, and means that there exists a mathematical proof that shows that *if* someone can break that implementation, *then* there is an efficient algorithm for a well known hard computational problem. Of course, if the cryptographic concept is not well-defined, then there can not exist such a mathematical proof (since the security property is not a well-defined mathematical statement). Thus having a well-defined cryptographic concept is a *necessary* condition for proven security.

Fuzzy security. Fuzzy security is actually the notion used by cryptographers throughout history until the last few decades. By *fuzzy security* I mean the following process: some guy comes up with some sort of cryptographic algorithm (let's think about an encryption scheme, but one can also think of other examples, such as hash functions or obfuscators). He then makes some vague claims about the security of this algorithm, and people start using it for applications of national or personal security of the utmost importance. Then, someone else (a hacker) manages to break this algorithm, usually with disastrous results to its users, and then the inventor or users either "tweak" the algorithm, hoping that the new tweak is secure, or one invent a new algorithm. The distinguishing mark of fuzzy security is *not* that it is often broken with disastrous outcomes. This is a side effect. The distinguishing mark is that there is never a rigorous definition of security, and so there is never a clearly stated conjecture of the security properties of this algorithm. Another common mark of fuzzy security is keeping the algorithm a secret. This is indeed unsurprising - if you don't know exactly what is the security of your algorithm, and you don't really understand what makes it secure, then keeping it secret seems like a good way to at least prolong the time it takes until a hacker finds a bug in it.

I want to stress that I do not intend to discredit the cryptographers throughout history that constructed "fuzzily secure" algorithms. Many of these people were geniuses with amazing intuitions, that paved the way for modern cryptography. However, this does not mean that we need to use their algorithms today. Today for most cryptographic tasks we do not need to use fuzzy security, since we have very good security definitions, and constructions that are reasonably conjectured to satisfy these definitions. One exception is indeed obfuscators, where so far no one has come up with a good definition for the security properties of obfuscators. Also (and this is not a coincidence) progress in obfuscator research seems to be of the sort mentioned above. One guy builds an obfusactor, an industry uses it, it gets broken, and then they build a new obfuscator (and/or try to pass laws making breaking obfuscators illegal..)

What is wrong with fuzzy security?

I'll answer that question with a question. What is wrong with fuzzy software engineering? By this I mean fuzzy specifications of regular (non-cryptographic) software modules. For example, would you use a function with the following specification in your code?

```
/*******************************************
 * Hopeful adder -                         *
 * Usually returns the sum of its two inputs. *
 *******************************************/
 int hopeAdd(int x, int y);
```

I wish to argue that it actually makes much *more* sense to use the hopeful adder function in your code, than to use a "fuzzily secure" cryptographic module. The reason is that you *can* test the normal-user inputs, but you *can not* test an hacker attack. By this I mean the following: if you use the hopeful adder in your code, and run it through many hours of beta-testing, and you never come across an input on which it fails, then it is indeed likely that this function will almost never fail when your application is used normally. However, if you use a "fuzzily-secure" cryptographic module in your application, and it does not get broken in the beta-test phase, this does not mean *anything* about the real-world security of your application. This is because a hacker will most likely attack your application in ways that were not predicted in the beta-testing phase, and will intentionally feed your application with the exact "weird" inputs that cause your module to fail. This is why in my view well-defined specifications are actually much *more* important in security than in other areas of software engineering. Of course, as all programmers know, using rigorously specified components does not guarantee that the overall system will be secure. However, using fuzzily specified components almost guarantees *insecurity*.

Aren't all systems insecure anyway? It's true that probably all large systems have security bugs, just as all large systems have other bugs as well. However, if one is working with well-defined secure components, it is possible *in principle* to build secure systems, just as it is possible *in principle* to build bug-free programs. The only problem is that it is very very difficult to build such "perfect" systems that are *large*. In spite of this, with time, and with repeated testing and scrutiny, systems can converge to that bug-free state (assuming that this time is indeed spent on fixing bugs and not on adding features - perhaps the best example of this is Knuth's TeX program). Such convergence cannot happen if one is using fuzzily secure components.

Can obfuscators enjoy well-defined security?

Indeed, this is the question that motivated the paper [1]. Our initial hope was that we can find a formal definition for software obfuscators, and a construction that can be proven to meet this definition (under some widely believed computational conjectures), and so bring obfuscator from the realm of fuzzy security, to the realm of well-defined (and proven) security. Unfortunately, throughout this research, whenever we came up with a definition, we eventually found a counterexample showing that this definition can not be met. The weakest natural definition for obfuscators we thought of was the following:

Definition 1: A compiler satisfying the functionality and efficency conditions is an *obfuscator* if for every program `P` that is hard to learn from its input/output behavior, a hacker can not reconstruct the source code of `P` from the obfuscated version of `P`.

The counterexample I mentioned above shows that Definition 1 is impossible to meet. This means that if one wants a well-defined and meaningful notion of obfuscation, one needs to come up with a new definition for obfuscators such that **(1)** the new definition is weak enough so that the existence of an obfuscator meeting it will not be contradicted immediately by our counterexample, but **(2)** the new definition is strong enough to provide some meaningful sense of security.

Now, the question of making obfuscators with well-defined security becomes the question of coming up with a definition satisfying these properties **(1)** and **(2)**. It seems to me that there are two natural ways to get such a definition:

- **Relaxing the hiding property:** A first approach may be to try to relax the definition of obfuscator by saying that an obfuscator does not need to hide all the details of the input program, but only some of them. However, this is already relaxed to the maximum by Definition 1, which says that the only thing that the obfuscator needs to do is to prevent the hacker from completely reconstructing the original source code.

- **Relaxing the class of programs:** a more promising direction is to say that the obfuscator only has to work for a particular *subclass* of programs. This subclass should have the following properties: **(1)** it should not contain the counterexample programs constructed by [1] but **(2)** it should contain the useful programs people want to obfuscate in practice. We discuss this direction in the paper. It's hard to find such a natural subclass, since it seems hard to distinguish between the counterexample we construct and between the useful programs people are interested in obfuscating.
In particular, some of our counterexamples can be obtained by relatively slight modifications to popular hash functions and private key encryption schemes. For example, we present in the paper a *pseudorandom* program family on which an obfuscator will fail completely. This means that when relaxing the class of programs, one must obtain a relaxation which contains the "natural" pseudorandom functions that people want to obfuscate, but does *not* contain "our" pseudorandom functions, even though the input/output behavior of "our" pseudorandom functions is indistinguishable from the input/output behavior of the "natural" pseudorandom function (since they are both pseudorandom). Note also that it must be easy to tell whether a program falls into this class, since otherwise a user won't be able to know if it is secure to obfuscate her program. In some intuitive sense, it seems that if you find such a class then you proved that some form of obfuscation does not exist...

Thus it seems that the paper [1] does present very serious challenges to anyone wishing to promote the state of obfuscators from its current "fuzzy security" level, to well-defined security. As a final note I want to mention that there are several interesting concepts related to obfuscators, and for many of them we have more open questions than results. The interested reader is referred to [1] for more details.

Are obfuscators useless?

I think that current obfuscators (or fuzzily secure components in general) may have limited uses. These are in cases when security *can* be tested, and a security breach can not cause catastrophic results. For example, in the setting of copyright protection, it is possible to detect "interesting" security breaks: If a hacker works alone and breaks the security to copy songs for her own use then this is undectable, but also causes only negligible damage to the record company. In contrast, to cause some damage, there must be an active "black market" of

copyrighted material, and such a market would be noticed by the studio, which would know that security has been broken. However, in order to be able to use this information, the system must be "planned to fail" in the sense that it should be easy to redeploy an alternative implementation, upgrade versions, etc.. This has not been the case in the past, for example in the DVD CSS algorithm. Thus, fuzzy security should be used only when one understands its inherent limitations. Of course, whenever possible, it is much better to use *well-defined* (and preferably *proven*) security.

- [1] B. Barak, O. Goldreich R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang, **On the (Im)possibility of Obfuscating Programs**, CRYPTO 2001
[abstract (html) | preliminary full version (ps) | Powerpoint XP presentation]

- [2] Slashdot discussion on the paper [1]. http://developers.slashdot.org/article.pl?sid=02/03/01/0344227