Random Oracles - OAEP

Anatoliy Gliberman, Dmitry Zontov, Patrick Nordahl

September 23, 2004

Reading Overview

There are two papers presented this week. The first paper, *Random Oracles are Practical: A Paradigm for Designing Efficient Protocols*, introduces a new methodology where cryptographic constructions are designed and proven in a "fantasy world" which makes the proofs of more efficient cryptosystems possible. The second paper, *Optimal Asymmetric Encryption – How to Encrypt with RSA*, introduces one such efficient construction, OAEP.

Paper 1

Random Oracles are Practical: A Paradigm for Designing Efficient Protocols

Section 1 starts by laying out the authors' agenda for the paper. At the time that this paper was written, cryptographic theorists were almost exclusively interested in issues that, while important for providing a formal foundation, had absolutely no use in the real world. The example given (building pseudorandom functions out of one-way functions) is just an exemplar of this foundational work and is not important to your understanding of this paper. Instead, the authors of this paper propose a new model, called the *random oracle model* that they claim more accurately models the real world while simultaneously making proofs easier.

Skip to section 2 to read about the notation for oracles, and then continue to 1.1

Section 2 introduces notation for this paper. The symbol Λ is an uppercase lambda. There should be an "of" in between class and negligible. The support, $[S]$, is the elements of the probability space with positive probability: in other words, the things that can actually happen. A *predicate* is a function that can return either true or false (in programming lingo, a predicate's return type is boolean). A random oracle is an *unkeyed* function (though not one that a computer can compute) that for each input, x, returns an infinitely long random string where each bit is chosen uniformly. However, it will always return the same random string when queried multiple times for the same x. They use the letter G to indicate an oracle that maps to the full random string (a generator), and H to indicate an oracle maps to a fixed length string (a hash function). To get multiple oracles from a single oracle, you can just say that the first few bits of the input determine which "virtual oracle" you're talking to. For example, let $G(x) = R(0||x)$ and $H(x) = R(1||x)$.[1] For now, just think of a trapdoor permutation as a regular pseudorandom permutation where we will give the adversary a function that computes f but not the function that computes f^{-1}. In their definition, d is the domain of the permutation. The security definition just says that an adversary who knows f but not f^{-1} and sees $y = f(x)$ isn't able to compute x. This is an older

[1]You'll also need to discarding the extra bits to make H a hash function.

definition of security (one-wayness) and we won't be using it in this class. Disregard the examples.

Section 1.1 begins with a good example between the "theoretical" and the "practical" view of primitives: at the time theoreticians were trying to show how to build pseudorandom functions from simple objects called one-way functions when in practice people were just using DES. What these authors call a protocol, we've been calling a construction or composition. Feel free to replace MD5 with SHA-1 if that makes you feel better about the claims.

The authors then introduce their methodology. Say we want to design a construction (in their terminology, protocol), P, that solves some problem "Π". Such a protocol is formally designed using a random oracle R (that the adversary also has access to). Once we've proved that P satisfies our definition of security, we wave our hands and replace the random oracle with some actual computable function h (which is usually a cryptographic hash function). We then (dubiously) claim that the construction is still secure. It's important to remember that this replacement is merely a heuristic — we haven't actually *proved* anything about the real construction. Notice that it is extremely important to choose h conservatively: h is already weaker than a random oracle, and any additional weaknesses make things even worse. For the remainder of this paper, the oracles are assumed to be instantiated by real-world hash functions such as MD5 or SHA-1.

Skip section 1.2 Much of this will be covered in the other paper.

Section 1.3 just talks about what led up to this work. It's not terribly relevant to your understanding of this paper.

Section 1.4 underlines one of the advantages of oracles – reduced complexity when it comes to proofs. We'll see this in action in the next paper

Skip everything up to section 6: Instantiation. This section talks about instantiating random oracles with primitives like hash functions. The examples given were earlier attacks on MD5 — with the newest results, MD5 should be avoided altogether in the instantiation of random oracles.

Paper 2:

Optimal Asymmetric Encryption – How to Encrypt with RSA

Section 1 identifies the goal of the paper as to perform asymmetric (public key) encryption that is provably secure. The following example considers a sender that possesses a k-bit to k-bit trapdoor permutation f (in other words $\{0,1\}^k$ to $\{0,1\}^k$), wants to transmit a message x to a receiver who holds the inverse permutation f^{-1}.

One type of function of interest to cryptologists is the "trap-door function." These are functions that are easy to calculate (in one direction), yet difficult to invert (calculate in the reverse direction)–like a trap door, easy to go down, difficult to come back. In this example the trapdoor permutation is a probabilistic function f defined for the domain d. There exists a certain function f^{-1} that takes in the output y of the trapdoor function that allows us to easily compute the "inverse" of f and get the initial string x. Without that inverse function it would be hard to compute the inverse just by having access to f and its output y. The notion of trapdoor permutation can be conveniently extended to the public-key encryption model. In a public encryption scheme we have two algorithms. The first algorithm is public (analogous to the trapdoor function f - that everyone has access to) that is capable of encrypting message x and producing an output ciphertext y. The second algorithm is private (analogous to the inverse trapdoor function f^{-1}) that allows us to decrypt the original message x from the ciphertext y.

Section 1.1 An important thing to realize in this section is that the G and H introduced here are actually random oracles as described in the previous paper.

The basic encryption scheme is fairly straightforward. The scheme makes use of two oracles G and H. Oracle G is represented by some random uniform mapping that takes inputs of length k_0 and outputs strings of length n. The second oracle is a hash function H with input of length n and the output of length k_0. The given scheme here is represented by a function f that is defined as follows: G takes a random string r to get $G(r)$. The output is then XOR'ed (preserves randomness) with the input string x. The result is then passed to H to get $H(xXORG(r))$ and the result is then XOR'with r and concatenated with the $G(r)$ that was previously XOR'ed with input string x.

Section 1.2 In this section a plaintext aware scheme is introduced. More about it will be discussed later in this paper. "Plaintext-awareness" implies that whenever an adversary creates a ciphertext, he must "know" its corresponding plaintext. As you might notice the main difference from the schemes described earlier, is that in terms of instantiation the input x is padded with $k1$ number of 0's (or any other fixed pattern for that matter) before the XOR logical operation. Again, we shall go into more detail as the paper progresses.

Section 1.3 The main idea here is that the authors are pitching to the "hands-on" practical scientists (that like the stuff that works as opposed to just "theory") by assuring them that the introduction of random oracles (hash functions) to the encryption scheme will not significantly affect its computational time or efficiency. RSA and modular squaring (also called Rabin encryption) are just two examples of trapdoor permutations, just like DES and AES were examples of PRPs. The constructions in this paper do not increase the number of times that f and f^{-1} need to be computed.

Section 1.4 In this paper we shall use the concept of a random oracles (ideal hash function) in order to perform proofs (refer to the previous paper).

Section 1.5 A concept of exact security is introduced. The idea behind it is that given the running time t, number of queries q_{gen} to G and q_{hash} to H, etc. we can come up with the exact probability for which this encryption scheme can be broken. Basically we attempt to assess the security of a given scheme by quantifying input variables. Given certain inputs, namely time and the number of queries to the function, we can predict the probability that a certain scheme can be broken.

Section 1.6 The assumption can be removed since we are assuming an idealistic hash function (random oracle).

Section 1.7 just shows some prior constructions. The most important thing to notice is how close they are to the final construction in this paper (OAEP) without being secure. It shows how tricky this stuff can be.

Section 2.1 This notation is much the same as that in the last paper.

Section 2.2 again talks about the notation for random oracles described earlier. It is an idealistic hash function whose output is restricted to a certain domain. The definition is similar as in the previous paper, the only distinction being that in previous the paper the domain for our oracles was infinite {0,1}* to {0,1} $^{\infty}$. In order to make such definition practical, we need to find a way to map the infinite random distribution to a finite domain. This is done by restricting the domain to $\{0{,}1\}^a$ to a certain number of bits a and then simply dropping the unused bits.

Section 2.3 brings us back to the trapdoor permutations. Here the scheme is described more formally in terms of notations. Going over the notation we get: a "trapdoor permutation generator" $\mathcal{F}$ generates two algorithms (f and its inverse, f^{-1}). An algorithm f will take an input string w that is a binary string of length k and produce an output string y. We now say that a certain adversary (algorithm M) that takes function f and its output y will have a certain success probability in coming up with original string w (tries to invert it) in at most t steps. The only difference between this definition and in one in section 2 of the previous paper is that we now introduce a restriction on a number of steps t and success probability value. Obviously we want to minimize such value.

Section 3.1 introduces an asymmetric encryption scheme. What you should get out of this section is that a generator $\mathcal{G}$ outputs two schemes (algorithms) $\mathcal{E}$, $\mathcal{D}$ and releases $\mathcal{E}$ to the public. Everyone has access to two independent oracles (hash functions) G and H. One encrypts message by using the $\mathcal{E}$ algorithm and a query to a public oracle G and H. In order to decrypt a message one will use the decryption algorithm $\mathcal{D}$ and again queries to oracles G and H. The most important limitation here is that a particular query is made to the oracle only once and the oracles will return the same output for a given identical input if queried a second time. Obviously every string that was encrypted using $\mathcal{E}$ would be possible to decrypt using $\mathcal{D}$. Also the decryption function $\mathcal{D}$ will give no output if the ciphertext is not valid.

Section 3.2 Semantic security captures the notion that you don't know anything after seeing the ciphertext that you didn't know beforehand.

Section 4 Same encryption schemes as introduced in section 1.1 and 3.1 more formally defined in terms of notations. As we have already mentioned earlier - $\mathcal{F}$ is a trapdoor permutation generator and $k_0(.)$ is a plaintext-length function. We run $\mathcal{F}$ to get (f and its inverse, $-f$). We now have two algorithms $\mathcal{E}$ (encryption algorithm - using encoding of f) and $\mathcal{G}$ (decryption algorithm - using encoding of f^{-1}. Both algorithms interact with two random oracles G and H.

The encryption algorithm $\mathcal{E}$ selects a random r of a certain length and then passes it to oracle $G(r)$. The result is XOR'ed with an input string x to get an output s. We then get an output t by passing in previously computed s to a random oracle H and then XOR'ing it with the random string r that was previously selected.

The two outputs s and t are then concatenated together to get a string w which will serve as an input to our trapdoor function f. The output of our encryption scheme is simply the output y from our trapdoor function f.

In order to decrypt the ciphertext y, we run the decryption algorithm $\mathcal{D}$ that will take in the output y and using the f^{-1} computes the input string w.

Section 5 We revisit the topic of "plaintext-awareness" one more time. Now we formally define the plaintext-aware encryption in terms of formal notation. Let $\mathcal{G}$ be a generator that provides us with two algorithms $\mathcal{E}$ (encryption algorithm) and $\mathcal{D}$ (decryption algorithm). Let us define Ω as a set of all oracles that provide us with two oracles G and H whose output is denoted as τ. We then let an adversary run his home-brewed algorithm B that has access to encryption scheme $\mathcal{E}$ and oracles to produce what an adversary claims to be a "valid" ciphertext y. The adversary also saves the output τ recieved from the oracles since the oracles return the same output for a duplicate query - there is no reason to query them again. We now introduce an algorithm K "the all-mighty knowledge extractor" that can tell if the ciphertext y produced by the adversary is indeed legit. This all-mighty algorithm has access to an encryption algorithm $\mathcal{E}$, adversary ciphertext y and the oracle query τ).

This "extractor" inverses the encryption from the ciphertext y that the adversary has provided us with, and gives us if the decrypted string x indeed matches the output of the real decryption algorithm $\mathcal{D}$. We claim that the probabilty of an adversary coming up with a valid ciphertext is minimal

Section 6 formally defines the plaintext-aware scheme (OAEP). As we have noted this scheme is similar to the basic encryption scheme with some additional padding. Now one might ask, "Why does the addition of padding make such an algorithm "plaintext-aware"?" The answer is as follows. We pad the input string with some fixed-bit pattern of k in length. It could be any fixed pattern (such as k 0's, a string "10101010101", etc.). We then proceed with the encryption scheme. Now let's say the adversary somehow came up with a ciphertext that he claims is "valid". All we have to do now is to run our decryption function and see if our message x has the necessary padding at the end intact.

We say that it is very hard for an adversary to come up with a "valid" ciphertext and leave the padding intact. This is very practical as we can guarantee with a certain amount of confidence that the encrypted message was not changed in transit (no need to rely other methods such as MAC's). Such an algorithm is very cheap to implement.

Notice, the last step of (2), if x has padding at the end (0^{k_0}) it is treated as a valid string and our ciphertext is valid. If not - nothing is returned (*).

Skip section 7.