PROFESSOR: All right, guys, let's get started. So today, we're going to talk about side-channel attacks, which is a general class of problems that comes up in all kinds of systems. Broadly, side-channel attacks are situations where you haven't thought about some information that your system might be revealing.

So typically, you have multiple components that you [INAUDIBLE] maybe a user talking to some server. And you're thinking, great, I know exactly all the bits going over some wire [INAUDIBLE] server, and those are secure. But it's often easy to miss some information revealed, either by user or by server. So the example that the paper for today talks about is a situation where the timing of the messages between the user and the server reveals some additional information that you wouldn't have otherwise learned by just observing the bits flowing between these two guys.

But In fact, there's a much broader class of side-channels you might worry about. Originally, side-channels showed up, or people discovered them in the '40s when they discovered that when you start typing characters on a teletype the electronics, or the electrical machinery in the teletype, would emit RF radiation. And you can hook up an oscilloscope nearby and just watch the characters being typed out by monitoring the frequency or RF frequencies that are going out of this machine. So RF radiation is a classic example of a side-channel that you might worry about.

And there's lots of examples lots of other examples that people have looked at, almost anything. So power usage is another side-channel you might worry about. So your computer is probably going to use different amounts of power depending on what exactly it's computing. I'm gonna go into other clever examples of sound turns out to also leak stuff.

There's a [? cute ?] paper that you can look at. The people listen to a printer and based on the sound the printer is making you can tell what characters it's printing. This is especially easy to do for dot matrix printers that make this very annoying sound when they're printing.

And in general, a good thing to think about, Kevin on Monday's lecture also mentioned some interesting side-channels that he's running through in his research. But, in particular, here we're going to look at the specific side-channel that David Brumley and Dan Boneh looked at in their paper-- I guess about 10 years ago now-- where they were able to extract a cryptographic key out of a web server running Apache by measuring the timing of different responses to different input packets from the adversarial client.

And in this particular case, they're going after a cryptographic key. In fact, many side-channel attacks target cryptographic keys partly because it's a little bit tricky to get lots of data through a side-channel. And cryptographic keys are one situation where getting a small number of bits helps you a lot. So in their attack they're able to extract maybe about 200 256 bits or so.

And just from those 200ish bits, they're able to break the cryptographic key of this web server. Whereas, if you're trying to leak some database full of Social Security numbers, then that'll be a lot of bits you have to leak to get out of this database. So that's why many of these side-channels, if you'll see them later on, they often focus on getting small secrets out, might be cryptographic keys or passwords. But in general, this is applicable to lots of other situations as well.

And one cool thing about this paper, before we jump into the details, is that they show that you actually do this over the network. So as you probably figured out from reading this paper, they have to do a lot of careful work to tease out these minute differences in timing information. So if you actually compute out the numbers from this paper, it turns out that each request that they sent to the server differs from potentially another [? website ?] by an order of 1 to 2 microseconds, which is pretty tiny.

So you have to be quite careful, and all of our network it might be hard to tell whether some server took 1 or 2 microseconds longer to process your request or not. And as a result, it was not so clear for whether you could mount this kind of attack over a very noisy network. And these guys were one of the first people to show that you can actually do this over a real ethernet network with a server sitting in one place, a client sitting somewhere else. And you could actually measure these differences partly by averaging, partly through other tricks. All right, does that make sense, the overall side-channel stuff?

All right. So the plan for the rest of this lecture is we'll first dive into the details of this RSA cryptosystem that these guys use. Then we'll not look at exactly why it's secure or not but we'll look at how do you implement it because that turns out to be critical for exploiting this particular side-channel. They carefully leverage various details of the implementation to figure out when there are some things faster or slower. And then we'll pop back out once we understand how RSA is implemented. Then we'll come back and figure out how do you attack it, how do you attack all these different organizations that RSA has. Sounds good? All right.

So I guess let's start off by looking at the high level plan for RSA. So RSA is a pretty widely used public key cryptosystem. We've mentioned these guys a couple of weeks ago in general in certificates, in the context of certificates. But now we're going to look at actually how it works. So typically there's 3 things you have to worry about. So there's generating a key, encrypting, and decrypting. So for RSA, the way you generate a key is you actually pick 2 large prime integers. So you're going to pick 2 primes, p and q.

And in the paper, these guys focus on p and q, which are about 512 bits each. So this is typically called 1,024 bit RSA because the resulting product of these primes that you're going to use in a second is a 1,000 bit integer number. These days, that's probably not a particularly good choice for the size of your RSA key because it makes it relatively easy for attackers to factor this-- not trivial but certainly viable. So if 10 years ago, this seemed like a potentially sensible parameter, now if you're

actually building a system, you should probably pick a 2,000 or 3,000 or even 4,000 bit RSA key. Well, that's what RSA key size means is the size of these primes.

And then, for convenience, we're going to talk about the number n, which is just the product of these 2 primes, p times q. All right. So now we know how to generate a key, now we need to figure out-- well this is at least part of a key-- now we're going to have to figure out how we're going to encrypt and decrypt messages. And the way we're going to encrypt and decrypt messages is by exponentiating numbers modulo this number n.

So it seems a little weird, but let's go with it for a second. So if you want to encrypt a message, then we're going to take a message m and transform it into m to the power e mod m. So e is going to be some exponent-- we'll talk about how to choose it in a second. But this is how we're going to encrypt a message.

We'll just take this message as an integer number and just exponentiate it. And then we'll see why this works in a second, but let's call this guy c, ciphertext. Then to decrypt it, we're going to somehow find an interesting other exponent where you can take a ciphertext c and if you exponentiate it to some power d mod m, then you'll magically get back the same message m. So this is the general plan: To encrypt, you exponentiate. To decrypt, you exponentiate by another exponent.

And in general, it seems a little hard to figure out how we're going to come up with these two magic numbers that somehow end up giving us back the same message. But it turns out that if you look at how exponentiation works or multiplication works, modulo of this number n. Then there's this cool property that if you have any number x, and you raise it to what's called a [? order ?] of phi function of n-- maybe I'll use more board space for this. This seems important.

So if you take x and you raise it to phi of n, then this is going to be equal to 1 mod m. And this phi function for our particular choice of n is pretty straightforward, it's actually p minus 1 times q minus 1. So this gives us hope that maybe if we pick ed so that e times d is 5n plus 1, then we're in good shape. Because then any message m we exponentiate it to e and d, we get back 1 times m because our ed

product is going to be roughly 5n plus 1, or maybe some constant alpha times 5n plus 1. Does this make sense? This is why the message is going to get decrypted correctly. And it turns out that there's a reasonably straightforward algorithm if you know this phi value for how to compute d given an e or e given a d. All right. Question.

AUDIENCE: Isn't 1 mod n just 1?

PROFESSOR: Yeah, so far we add one more. Sorry?

AUDIENCE: Like, up over there.

PROFESSOR: Yeah, this one?

AUDIENCE: Yeah.

PROFESSOR: Isn't 1 mod n just 1? Sorry, I mean this. So when I say this 1 n, it means that both sides taken 1n are equal. So what this means is if you want to think of mod as literally an operator, you would write this guy mod m equals 1 mod m. So that's what mod m on the side means. Like, the whole equality is mod m. Sorry for the [INAUDIBLE]. Make sense? All right.

So what this basically means for RSA is that we're going to pick some value e. So e is going to be our encryption value. And then from e we're going to generate d to be basically 1 over e mod phi of n. And there's some Euclidean algorithms you can use to do this computation efficiently. But in order to do this you actually have to know this phi of n, which requires knowing the factorization of our number n into p and q.

All right. So finally, RSA ends up being a system where the public key is this number n and this encryption exponent e. So n and e are public, and d should be private. So then anyone can exponentiate a message to encrypt it for you. But only you know this value d and therefore can decrypt messages. And as long as you don't know this factorization of p and q, of n to p and q, then you don't know what this [? phi del ?] is. And as a result, it's actually difficult to compute this d value. So this is roughly what RSA is. High level. Does this make sense? All right.

So there's 2 things I want to talk about now that we at least have the basic [? implementation ?] for RSA. There's tricks to use it correctly and pitfalls and how to use RSA. And then there's all kinds of implementation tricks on how do you actually implement [? root ?] code to do these exponentiations and do them efficiently. There's actually more trivial because these are all large numbers, these are 1,000 bit integers that can't just do a multiply instruction for. Probably going to take a fair amount of time to do these operations. All right.

So the first thing I want to mention is the various RSA pitfalls. One of them we're actually going to rely on in a little bit. One property is, that it's multiplicative. So what I mean by this is that suppose we have 2 messages. Suppose we have m0 and m1. And suppose I encrypt these guys, so I encrypt m0, I'm going to get m0 to the power e mod n. And if I encrypt m1, then I'd get m1 to the e mod n. The problem is-- not necessarily a problem but could be a surprise to someone using RSA-- it's very easy to generate an encryption of m0 times m1 because you just multiply these 2 numbers. If you multiply these guys out, you're going to get m0 m1 to the e mod n.

This is a correct encryption under this simplistic use of RSA for the value m0 times m1. I mean at this point, it's not a huge problem because if you aren't able to decrypt it, you're just able to construct this encrypted message. But it might be that the overall system maybe allows you to decrypt certain messages. And if it allows you to decrypt this message that you construct yourself, maybe you can now go back and figure out what are these messages. So it's maybe not a great plan to be ignorant of this fact. This has certainly come back to bite a number of protocols that use RSA. There's one property, we'll actually use it as a defensive mechanism towards the end of the lecture.

Another property of RSA that you probably want to watch out for is the fact that it's deterministic. So in this [? naive ?] implementation that I just described here, if you take a message m and you encrypt it, you're going to get m to the e mod n, which is a deterministic function of the message. So if you encrypt it again, you'll get exactly the same encryption.

This is not surprising but it might not be a desirable property because if I see you send send some message encrypted with RSA, and I want to know what it is, it might be hard for me to decrypt it. But I can try different things and I can see, well are you sending this message? I'll encrypt it and see if you get the same ciphertext. And if so, then I'll know that's what you encrypted. Because all I need to encrypt a message is the publicly known public key, which is n and the number e. So that's not so great. And you might want to watch out for this property if you're actually using RSA. So all of these [? primitives are ?] probably a little bit hard to use directly.

What people do in practice in order to avoid these problems with RSA is they encode the message in a certain way before encrypting it. Instead of directly exponentiating a message, it actually takes some function of a message, and then they encrypt that. mod n. And this function f, the right one to use these days, is probably something called optimal asymmetric encryption padding, O A E P. You can look it up. It's something coded that has two interesting properties.

First of all, it injects randomness. You can think of f of n as generating 1,000 bit message that you're going to encrypt. Part of this message is going to be your message m in the middle here. So that you can get it back when you decrypt, of course. [INAUDIBLE]. So there's 2 interesting things you want to do. You want to put in some randomness here, some value r so that when you encrypt the message multiple times, you'll get different results out of each time so then it's not deterministic anymore.

And in order to defeat this multiplicative property and other kinds of problems, you're going to put in some fixed padding here. You can think of this as an altering sequence of 1 0 1 0 1 0. You can do better things. But roughly it's some predictable sequence that you put in here and whenever you decrypt, you make sure the sequence is still there. Even in multiplication it's going to destroy this bit power. And then you should be clear that someone tampered with my message and reject it. And if it's still there, then presumably, sometimes provably, no one tampered with your message, and as a result you should be able to accept it. And treat message

m as correctly encrypted by someone. Make sense? Yeah?

AUDIENCE: If the attacker knows how big the pad is, can't they put a 1 in the lowest place and then [INAUDIBLE] under multiplication?

PROFESSOR: Yeah, maybe. It's a little bit tricky because this randomness is going to bleed over. So the particular construction of this O A E P is a little bit more sophisticated than this. But if you imagine this is integer multiplication not bit-wise multiplication. And so this randomness is going to bleed over somewhere, and you can construct O A E P scheme such that this doesn't happen. [INAUDIBLE] Make sense? All right.

So it turns out that basically you shouldn't really use this RSA math directly, you should use some library in practice that implements all those things correctly for you. And use it just as an encrypt/decrypt parameter. But it turns out these details will come in and matter for us because we're actually trying to figure out how to break or how to attack an existing RSA implementation.

So in particular the attack from this paper is going to exploit the fact that the server is going to check for this padding when they get a message. So this is how we're going to time how long it takes a server to decrypt. We're going to send some random message, or some carefully constructed message. But the message wasn't constructed by taking a real m and encrypting it.

We're going to construct a careful ciphertext integer value. And the server is going to decrypt it, it's going to decrypt to some nonsense, and the padding is going to not match with a very high probability. And immediately the server is going to reject it. And the reason this is going to be good for us is because it will tell us exactly how long it took the server to get to this point, just do the RSA decryption, get this message, check the padding, and reject it. So that's what we're going to be measuring in this attack from the paper. Does that make sense? So there's some integrity component to the the message that allows us to time the decryption leading up to it. All right.

So now let's talk about how to do you actually implement RSA. So the core of it is

really this exponentiation, which is not exactly trivial to do as I was mentioning earlier because all these numbers are very large integers. So the message itself is going to be at least, in this paper, 1,000 bit integer. And the exponent itself is also going to be pretty large.

The encryption exponent is at least well known. But the decryption exponent better be also a large integer also on the order of 1,000 bits. So you have a 1,000 bit integer you want to exponentiate to another 1,000 bit integer power modulo some other 1,000 bit integer n that's going to be a little messy, if you just do [? the naive thing. ?] So almost everyone has lots of optimizations in their RSA implementations to make this go a little bit faster.

And there's four optimizations that matter for the purpose of this attack. There is actually more tricks that you can play, but the most important ones are these. So first there's something called the Chinese remainder theorem, or C R T. And just to remind you from grade school or high school maybe what this remainder theorem says.

It actually says that if you have two numbers and you have some value x and you know that x is equal to a1 mod p. And you know that x is equal to a2 mod q, where p and q are prime numbers. And this modular equality applies to the whole equation. Then it turns out that there's a unique solution to this is mod p q. So there's are some x equals to some x prime mod pq. And in fact, there's a unique such x prime, and it's actually very efficient to compute. So the Chinese remainder theorem also comes with an algorithm for how to compute this unique x prime that's equal to x mod pq given the values a1 and a2 mod p and q, respectively. Make sense?

OK, so how can you use this Chinese remainder theorem to speed up modular exponentiation? So the way this is going to help us is that if you notice all the time we're doing this computational of some bunch of stuff modulo n, which is p times q. And the Chinese remainder theorem says that if you want the value of something mod p times q, it suffices to compute the value of that thing mod p and the value of

that thing mod q. And then use the Chinese remainder theorem to figure out the unique solution to what this thing is mod p times q. All right, why is this faster? Seems like you're basically doing the same thing twice, and that's more work to recombine it Is this going to save me anything? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, they're certainly smaller, they're not that smaller. And so p and q, so n is 1,000 bits, p and q are both 500 bits, they're not quite to the machine word size yet. But it is going to help us because most of the stuff we're doing in this computation is all these multiplications. And roughly multiplication is quadratic in the size of the thing you're multiplying because the grade school method of multiplication you take all the digits and multiply them by all the other digits in the number.

And as a result, doing exponentiation multiplication is roughly quadratic in the input side. So if we shrink the value of p, we basically go from 1,000 bits to 512 bits, we reduce the size of our input by 2. So this means all this multiplication exponentiation is going to be roughly 4 times cheaper. So even though we do it twice, each time is 4 times faster. So overall, the CRT optimization is going to give us basically a 2x performance boost for doing any RSA operation both, in the encryption and decryption side. That make sense? All right. So that's the first optimization that most people use.

The second thing that most implementations do is a technique called sliding windows. And we'll look at this implementation in 2 steps so this implementation is going to be concerned with what basic operations are going to perform to do this exponentiation. Suppose you have some ciphertext c that's now 500 bits because you were not doing mod p or mod q. We have a 500 bit c and, similarly, roughly a 500 bit d as well.

So how do we raise c to the power d? I guess the stupid way that is to take c and keep multiplying d times. But d is very big, it's 2 to the 500. So that's never going to finish. So a more amenable, or more performant, plan is to do what's called repeat of squaring. So that's the step before sliding windows.

So this technique called repeated squaring looks like this. So if you want to compute c to the power 2 x, then you can actually compute c to the x and then square it. So in our naive plan, computing c to the 2x would have involved us making twice as many iterations of multiplying because it's multiplying c twice many times. But in fact, you could be clever and just compute c to the x and then square it later. So this works well, and this means that if you're computing c to some even exponent, this works. And conversely, if you're computing c to some 2x plus 1, then you could imagine this is just c to the x squared times another c. So this is what's called repeated squaring.

And this now allows us to compute these exponentiations, or modular exponentiations, in a time that's basically linear in the size of the exponent. So for every bit in the exponent, we're going to either square something or square something then do an extra multiplication. So that's the plan for repeated squaring. So now we can at least have non-embarrassing run times for computing modular exponents. Does this make sense, why this is working and why it's faster?

All right, so what's this sliding windows trick that the paper talks about? So this is a little bit more sophisticated than this repeating squaring business. And basically the squaring is going to be pretty much inevitable. But what the sliding windows optimization is trying do is reduce the overhead of multiplying by this extra c down here.

So suppose if you have some number that has several 1 bits in the exponent, for every 1 bit in the exponent in the binder of presentation, you're going to have do this step instead of this step. Because for every odd number, you're going to have to multiply by c. So these guys would like to not multiply by this c as often.

So the plan is to precompute different powers of c. So what we're going to do is we're going to generate a table that says, well, here's the value of c to the x-- sorry, c to the 1-- here's the value of c to the 3, c to the 7. And I think [? in open ?] as a cell, it goes up to c to the 31st. So this table is going to just be precomputed when you want to do some modular exponentiation. You're going to precompute all the

slots in this table. And then when you want to do this exponentiation, instead of doing the repeated squaring and multiplying by this c every time,

You're going to use a different formula. It says as well if you have c to the 32x plus some y, well you can do c to the x, and you can do repeated squaring-- very much like before-- this is to get the 32, there's like 5 powers of 2 here times c to the y. And c to the y, you can get out of this table. So you can see that we're doing the same number of squaring as before here. But we don't have to multiply by c as many times. You're going to fish it out of this table and do several multiplies by c for the cost of a single multiply. This make sense? Yeah?

AUDIENCE: How do you determine x and y in the first place?

PROFESSOR: How do determine y?

AUDIENCE: X and y.

PROFESSOR: Oh, OK. So let's look at that. So for repeated squaring, well actually in both cases, what you want to do is you want to look at the exponent that you're trying to use in a binary representation. So suppose I'm trying to compute the value of c to the exponent, I don't know, 1 0 1 1 0 1 0, and maybe there's more bits. OK, so if we wanted to do repeated squaring, then you look at the lowest bit here-- it's 0. So what you're going to write down is this is equal to c to the 1 0 1 1 0 1 squared.

OK, so now if only you knew this value, then you could just square it. OK, now we're going to compute this guy, so c to the 1 0 1 1 0 1 is equal to-- well here we can't use this rule because it's not 2x-- it's going to be to the x plus 1. So now we're going to write this is c to the 1 0 1 1 0 squared times another c. Because it's this prefix times 2 plus this one of m. That's how you fish it out for repeated squaring.

And for sliding window, you just grab more bits from the low end. So if you wanted to do the sliding window trick here instead of taking one c out, suppose we do-- instead of this giant table-- maybe we do 3 bits at a time. So we go off to c to the 7th. So here you would grab the first 3 bits, and that's what you would compute here: c to the 1 0 1 to the 8th power. And then, the rest is c to the 1 0 1 power here.

It's a little unfortunate these are the same thing, but really there's more bits here. But here, this is the thing that you're going to look up in the table. This is c to the 5th in decimal. And this says you're going to keep doing the sliding window to compute this value. Make sense?

This just saves on how many times you have to multiply by c by pre-multiplying it a bunch of times. [? And the cell guys ?] at least 10 years ago thought that going up to 32 power was the best plan in terms of efficiency because there's some trade off here, right? You spend time preconfiguring this table, but then if this table is too giant, you're not going to use some entries, because if you run this table out to, I don't know, c to the 128 but you're computing just like 500 [? full bit ?] exponents, maybe you're not going to use all these entries. So it's gonna be a waste of time. Question.

AUDIENCE: [INAUDIBLE] Is there a reason not to compute the table [INAUDIBLE]? [INAUDIBLE].

PROFESSOR: It ends up being the case that you don't want to-- well there's two things going on. One is that you'll have now code to check whether the entry is filled in or not, and that'll probably reduce your branch predictor accuracy on the CPU So it will run slower in the common case because if you [INAUDIBLE] with the entries there.

Another slightly annoying thing is that it turns out this entry leaks stuff through a different side-channel, namely cache access patterns. So if you have some other process on the same CPU, you can sort of see which cache addresses are getting evicted out of the cache or are slower because someone accessed this entry or this entry. And the bigger this table gets, the easier it is to tell what the exponent bits were.

In the limit, this table is gigantic and just telling, just being able to tell which cache address on this CPU had a [? miss ?] tells you that the encryption process must have accessed that entry in the table. And tells you that, oh that long bit sequence appears somewhere in your secret key exponent. So I guess the answer isn't mathematically you could totally fill this in on demand.

In practice, you probably don't want it to be that giant. And also, if you have it's particularly giant, you aren't going to be able to use entries as efficiently as well. You can reuse these entries as you're computing. [INAUDIBLE] It's not actually that expensive because you use c to the cubed when you're computing c to the 7th and so on and so forth. It's not that bad. Make sense? Other questions? All right.

So this is the repeated squaring and sliding window optimization that open [? a cell ?] implements [INAUDIBLE] I don't actually know whether they still have the same size of the sliding window or not. But it does actually give you a fair bit of speed up. So before you had to square for every bit in the exponent.

And then you'd have to have a multiply for every 1 bit. So if you have a 500 bit exponent then you're going to do 500 squarings and, on average, roughly 256 multiplications by c. So with sliding windows, you're going to still do the 512 squarings because there's no getting around that. But instead of doing 256 multiplies by c, you're going to hopefully do way fewer, maybe something on the order of 32 [INAUDIBLE] multiplies by some entry in this table. So that's the general plan. [INAUDIBLE] Not as dramatic as CRT, not 2x, but it could save you like almost 1.5x. All depending on exactly what [INAUDIBLE]. Make sense? Another question about this? All right.

So these are the [? roughly ?] easier optimizations. And then there's two clever tricks playing with numbers for how to do just a multiplication more efficiently. So the first one of these optimizations that we're going to look at-- I think I'll raise this board-- is called this Montgomery representation. And we'll see in a second why it's particularly important for us.

So the problem that this Montgomery representation optimization is trying to solve for us is the fact that every time we do a multiply, we get a number that keeps growing and growing and growing. In particular, both in sliding windows or in repeated squaring, actually when you square you multiply 2 numbers together, when you multiply by c to the y, you multiply 2 numbers together.

And the problem is that if the inputs to the multiplication were, let's say, 512 bits each. Then the result of the multiplication is going to be 1,000 bits. And then you'd take this 1,000 bit result and you multiply it again by something like five [INAUDIBLE] bits. And now it's 1,500 bits, 2,000 bits, 2,500 bits, and it keeps growing and growing.

And you really don't want this because multiplications [? quadratic ?] in the size of the number we're multiplying. So we have to keep the size of our number as small as possible, which means basically 512 bits because all this computation is mod p or mod q. Yeah?

AUDIENCE: What do you want [INAUDIBLE]?

PROFESSOR: That's right, yeah. So the cool thing is that we can keep this number down because what we do is, let's say, we want to compute c to the x just for this example. Squared. Squared again. Squared again. What you could do is you compute c to the x then you take mod p, let's say, right. Then you square it then you do mod p again. Then you square it again, and then you do mod p again. And so on.

So this is basically what you're proposing. So this is great. In fact, this keeps it size of our numbers to basically five total bits, which is about as small as we can get. This is good in terms of keeping down the size of these numbers for multiplication. But it's actually kind of expensive to do this mod p operation. Because the way that you do mod p something is you basically have to do division. And division is way worse than multiplication.

I'm not going to go through the algorithms for division, but it's really slow. You usually want to avoid division as much as possible. Because it's not even just a straightforward programming thing, you have to do some approximation algorithm, some sort of Newton's method of some sort and just keep it [INAUDIBLE]. It's going to be slow.

And in the main implementation, this actually turns out to be the slowest part of doing multiplication. The multiplication is cheap. But then doing mod p or mod q to

bring it back down in size is going to be actually more expensive than the multiplying. So that's actually kind of a bummer.

So the way that we're going to get around this is by doing this multiplication, this clever other representation, and also I'll show you the trick here. Let's see. Bear with me for a second, and then we'll and then see why it's so fast to use this Montgomery trick.

And the basic idea is to represent numbers, these are regular numbers that you might actually want to multiply. And we're going to have a different representation for these numbers, called the Montgomery representation. And that representation is actually very easy. We just take the value a and we multiply it by some magic value R.

I'll tell you what this R is in a second. But let's first figure out if you pick some arbitrary value R, what's going to happen here? So we take 2 numbers, a and b. Their Montgomery representations are sort of expectedly. A is aR, b is bR.

And if you want to compute the product of a times b, well in Montgomery space, you can also multiply these guys out. You can take aR multiply it by bR. And what you get here is ab times R squared. So there are two Rs now. That's kind of annoying, but you can divide that by R. And we get ab times R. So this is probably weird in a sense that why would you multiply this extra number. But let's first figure out whether this is correct. And then we'll figure out why this is going to be faster.

So it's correct in the sense that it's very easy. If you want to multiply some numbers, we just multiply by this R value and get the Montgomery representation. Then we can do all these multiplications to these Montgomery forms. And every time we multiply 2 numbers, we have to divide by R, look at the Montgomery form of the multiplication result. And then when we're done doing all of our squarings, multiplication, all this stuff, we're going to move back to the normal, regular form by just dividing by R one last time.

AUDIENCE: [INAUDIBLE]

PROFESSOR: We're now going to pick R to be a very nice number. And in particular, we're going to pick R to be a very nice number to make this division by R very fast. And the cool thing is that if this division by R is going to be very fast, then this is going to be a small number and we're not going to have to do this mod q very often. In particular, aR, let's say, is also going to be roughly 500 bits because it's all actually mod p or mod q. So aR is 500 bits.

BR is going to also be 500 bits. So this product is going to be 1,000 bits. This R is going to be this nice 500 roughly bit number, same size as p. And if we can make this division to be fast, then the result is going to be a roughly 500 bit number here. So we were able to do the multiplying without having to do an extra divide. Dividing by R cheaply gives us this small result, getting us out of doing a mod p for most situations.

OK, so what is this weird number that I keep talking about? Well R is just going to be 2 to 512. It's going to be 1 followed by a ton of zeros. So multiplying by this is easy, you just append a bunch of zeros to a number. Dividing could be easy if the low bits of the result are all zeros. So if you have a value that's a bunch of bits followed by 512 zeros, then dividing by 2 to the 512 is cheap. You just discard the zeros on the right-hand side. And that's actually the correct division. Does that make sense?

The slight problem is that we actually don't have zeros on the right hand side when you do this multiplication. These are like real 512 bit numbers with all the 512 bits used. So this will be a 1,000 bit number [? or ?] with all this bits also set to randomly 0 or 1, depending on what's going on. So we can't just discard the low bits.

But the cleverness comes from the fact that the only thing we care about is the value of this thing mod p. So you can always add multiples of p to this value without changing it when it's equivalent to mod p. And as a result, we can add multiples of p to get the low bits to all be zeros.

So let's look through some simple examples. I'm not going to write out 512 bits on the board. But suppose that-- here's a short example. Suppose that we have a situation where our value R is 2 to the 4th. So it's 1 followed by four zeros. So this is

a much smaller example than the real thing. But let's see how this Montgomery division is going to work out. So suppose we're going to try to compute stuff mod q, where q, let's say, is maybe 7. So this is 1 1 1 in binary form. And what we're going to try to do is maybe we did some multiplication. And this value aR times bR is equal to this binary presentation 1 1 0 1 0. So this is going to be the value of aR times bR.

How do we divide it by R? So clearly the low four bits aren't all 0, so we can't just divide it out. But we can add multiples of q. In particular, we can add 2 times q. So 2q is equal to 1 1 1 0. And now what we get is 0 0, carry a 1, 0, carry a 1, 1, carry a 1, 0 1. I hope I did that right. So this is what we get. So now we get aR bR plus 2 cubed. But we actually don't care about the plus 2 cubed. It's actually fine because all we care about is the value of mod q.

And now we're closer, we have three 0 bits at the bottom. Now we can add another multiple of q. This time it's going to be probably 8q. So we add 1 1 1 here 0 0. And if we add it, we're going to get, let's say, 0 0 0 then add these two guys 0, carry a 1, 0, carry a 1, 1 1. I think that's right. But now we have our original aR bR plus 2q plus 8q is equal to this thing. And finally, we can divide this thing by R very cheaply. Because we just discard the low four zeros. Make sense? Question.

AUDIENCE: Is aR bR always going to end in, I guess, 1,024 zeros?

PROFESSOR: No, and the reason is that-- OK, here is the thing that's maybe confusing. A was, let's say, 512 bits. Then you multiply it by R. So here, you're right. This value is that 1,000 bit number where the high bit is a, the high 512 bits are a. And the low bits are all zeros. But then, you're going [? to do it with ?] mod q to bring it down to make it smaller. And in general, this is going to be the case. Because [? it only ?] has these low zeros the first time you convert it. But after you do a couple multiplications, they're going to be arbitrary bits. So these guys are-- so I really should have written mod q here-- and to compute this mod q as soon as you do the conversion to keep the whole value small.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, so the initial conversion is expensive or at least it's as expensive as doing a regular modulus during the multiplication. The cool thing is that you pay this cost just once when you do the conversion into Montgomery form. And then, instead of converting it back at every step, you just keep it in Montgomery form.

But remember that in order to do an exponentiation to an exponent which has 512 bits, you're saying you're going to have to do over 500 multiplications because we have to do at least 500 squarings plus then some. So you do these mod q twice and then you get a lot of cheap divisions if you stay in this form. And then you do a division by R to get back to this form again. So instead of doing 500 mod qs for every multiplication step, you do it twice mod q. And then you keep doing these divisions by R cheaply using this trick. Question.

AUDIENCE: So when you're adding the multiples of q and then dividing by R, [INAUDIBLE]

PROFESSOR: Because it's actually mod q means the remainder when you divide by q. So x plus y times q, mod q is just x.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So in this case, dividing by-- so another sort of nice property is that because it's all modulus at prime number-- it's also true that if you have x plus yq divided by R, mod q is actually the same as x divided by R mod q. The way to think of it is that there's no real division in modular arithmetic. It's just an inverse. So what this really says is this is actually x plus yq times some number called R inverse. And then you compute this whole thing mod q. And then you could think of this as x times R inverse mod q plus y [? u ?] R inverse mod q. And this thing cancels out because it's something times q.

And there's some closed form for this thing. So here I did it by bit by bit, 2q then 8q, et cetera. It's actually a nice closed formula you can compute-- it's in the lecture notes, but it's probably not worth spending time on the board here-- for how do you figure out what multiple of q should you add to get all the low bits to turn to 0. So then it turns out that in order to do this division by R, you just need to compute this

magic multiple of q, add it. And then discard the low bits and that brings your number back to 512 bits, or whatever the size is.

OK. And here's the subtlety. The only reason we're talking about this is that there's something funny going on here that is going to allow us to learn timing information. And in particular, even though we divided by R, we know the result is going to be 512 bits. But it still might be greater than q because q isn't exactly [? up to 512 ?], it's not a 512 bit number. So it might be a little bit less than R. So it might be that after we do this cheap division by R, [? the way ?] we subtract out q one more time because we get something that's small but not quite small enough.

So there's a chance that after doing this division, we maybe have to also subtract q again. And this subtraction is going to be part of what this attack is all about. It turns out that subtracting this q adds time. And someone figured out-- not these guys but some previous work-- that you show that this probability of doing this thing, this is called an extractor reduction. This probability sort of depends on the particular value that you're exponentiating. So if you're computing x to the d mod q, the probability of an extra reduction, at some point while computing x to the d mod q, is going to be equal to x mod q divided by 2R.

So if we're going to be computing x to the mod q, then depending on what the value of x mod q is, whether it's big or small, you're going to have even more or less of these extra reductions. And just to show you where this is going to fit in, this is actually going to happen in the decrypt step, because during the decrypt step, the server is going to be computing c to the d. And this says the extractor reductions are going to be proportional to how close x, or c in this case, is to the value q.

So this is going to be worrisome, right, because the attacker gets to choose the input c. And the number of extractor reductions is going to be proportional to how close the c is to one of the factors, the q. And this is how you're going to tell I'm getting close to the q, or I've overshot q. And all of a sudden, there's no extractor reductions, it's probably because x mod q is very small the x is q plus little epsilon. And it's very small. So that's one part of the timing attack we're going to be looking

at in a second. I don't have any proof that this actually true [INAUDIBLE] these extractor reductions work like this. Yea, question.

AUDIENCE: What happens if you don't do this extra reduction?

PROFESSOR: Oh, what happens if you don't do this extractor reduction? You can avoid this extra reduction. And then you just have to do some extra probably modular reductions later. I think the math just works out nicely this way for the Montgomery form. I think for many of these things it's actually once you look at them as a timing channel [INAUDIBLE] [? think ?] don't do this at all, or maybe you should do some other plan. So you're right,

I think you could probably avoid this extra reduction and probably just do the mod q, perhaps at the end. I haven't actually tried implementing this. But it seems like it could work. It might be that you just have to do mod q once [? there ?], which you'll probably have to do anyway. So it's not super clear. Maybe it's [INAUDIBLE] probably not q.

So in light of the fact that [INAUDIBLE]. Actually, I shouldn't speak authoritatively to this. I haven't tired implementing this. So maybe there's some deep reason why this extractor reduction has to happen. I couldn't think of one. All right, questions?

So here's the last piece of the puzzle for how OpenSSL, this library that this paper attacks implements multiplication. So this Montgomery trick is great for avoiding the mod q part during modular multiplication. But then there's a question of how do you actually multiply two numbers together. So we're doing lower and lower level.

So suppose you have [? the raw ?] multiplication. So this is not even modular multiplication. You have two numbers, a and b. And both these guys are 512 bit numbers. How do you multiply them together when your machine is only a 32 bit machine, like the guys in the paper, or a 64 bit, but still, same thing? How would you implement multiplication of these guys? Any suggestions?

Well I guess it was a straightforward question, you just represent a and b as a sequence of machine [? words. ?] And then you just do this quadratic product of

these two guys. [INAUDIBLE] see a simple example, instead of thinking of a 512 bit number, let's think of these guys as 64 bit numbers and we're on a 32 bit machine. Right. So we're going to have values. The value of a is going to be represented by two [? very ?] different things.

It's going to be, let's call it, a1 and a0. So a0 is the low bit, a1 is the high bit. And similarly, we're going to represent b as two things, b1 b0. So then a naive way to represent a b is going to be to multiply all these guys out. So it's going to be a three cell number. The high bit is going to be a1 b1. The low bit is going to be a0 b0. And the middle word is going to be a1 b0 plus a0 b1. So this is how you do the multiplication, right. Question?

AUDIENCE: So I was going to say are you using [INAUDIBLE] method?

PROFESSOR: Yeah, so this is like a clever method alternative for doing multiplication, which doesn't involve four steps. Here, you have to do four multiplications. There's this clever other method, Karatsuba. Do they teach this in 601 or something these days?

AUDIENCE: 042.

PROFESSOR: 042, excellent. Yeah, that's a very nice method. Almost every cryptographic library implements this. And for those of you that, I guess, weren't undergrads here, since we have grad students maybe they haven't seen Karatsuba. I'll just write it out on the board. It's a clever thing the first time you see it. And what you can do is basically compute out three values. You're going to compute out a1 b1. You're going to also compute a1 minus b0 times b1 minus-- sorry-- a1 minus a0, b1 minus b0. And a0 b0. And this does three multiplications instead of four. And it turns out you can actually reconstruct this value from these three multiplication results.

And the particular way to do it is this is going to be the-- let me write it out in a different form. So we're going to have 2 to the 64 times-- sorry-- 2 to the 64 plus 2 to the 32 times a1 b1 plus 2 to the 32 times minus that little guy in the middle a1 minus a0 b1 minus b0.

And finally, we're going to do 2 to the 32 plus 1 times a0 b0. And it's a little messy, but actually if you work through the details, you'll end up convincing yourself hopefully that this value is exactly the same as this value. So it's a clever. But nonetheless, it saves you one multiplication. And the way we apply this to doing much larger multiplications is that you recursively keep going down.

So if you have 512 bit values, you could break it down to 256 bit multiplication. You do three 256 bit multiplications. And then each of those you're going to do using the same Karatsuba trick recursively. And eventually you'll get down to machine size, which you can just do with a single machine instruction. [INAUDIBLE] This make sense?

So what's the timing attack here? How do these guys exploit this Karatsuba multiplication? Well, it turns out that OpenSSL worries about basically two kinds of multiplications that you might need to do. One is a multiplication between two large numbers that are about the same size. So this happens a lot when we're doing this modular exponentiation because all the values we're going to be multiplying are all going to be roughly 512 bits in size.

So when we're multiplying by c to the y or doing a squaring, we're multiplying two things that are about the same size. And then this Karatsuba trick makes a lot of sense because, instead of computing stuff in times squared of the input size, Karatsuba is roughly n to the 1.58, something like that. So it's much faster.

But then there's this other situation where OpenSSL might be multiplying two numbers that are very different in size: one that's very big, and one that's very small. And in that case you could use Karatsuba, but then it's going to get you slower than doing the naive thing. Suppose you're trying to multiply a 512 bit number by a 64 bit number, you'd rather just do the straightforward thing, where you just multiply by each of the things in the 64 bit number plus 2n instead of n to the 1.58 something.

So as a result, the OpenSSL guys tried to be clever, and that's where often problems start. They decided that they'll actually switch dynamically between this

Karatsuba efficient thing and this sort of grade school method of multiplication here. And their heuristic was basically if the two things you're multiplying are exactly the same number of machine words, so they at least have the same number of bits up to 32-bit units, then they'll go to Karatsuba. And if the two things they're multiplying have a different number or 32 bit units, then they'll do the quadratic or straightforward or regular, normal multiplication.

And there you can see if your number all of a sudden switches to be a little bit smaller, then you're going to switch from the sufficient thing to this other multiplication method. And presumably, the cutoff point isn't going to be exactly smooth so you'll be able to tell all of a sudden, it's now taking a lot longer to multiply or a lot shorter to multiply than before. And that's what these guys exploit in their timing attack again. Does that make sense? What's going on with the [INAUDIBLE] All right.

So I think I'm now done with telling you about all the weird implementation tricks that people play when implementing RSA in practice. So now let's try to put them back together into an entire web server and figure out how do you [? tickle ?] all these interesting bits of the implementation from the input network packet.

So what happens in a web server is that the web server, if you remember from the HTTPS lecture, has a secret key. And it uses the secret key to prove that it's the correct owner of all that certificate in the HTTPS protocol or in TLS. And they way this works is that the clients send some randomly chosen bits, and the bits are encrypted using the server's public key. And the server in this TLS protocol decrypts this message. And if the message checks out, it uses those random bits to establish a [? session ?]. But in this case, the message isn't going to check out. The message is going to be carefully chosen, the padding bits aren't going to match, and the server is going to return error as soon as it finishes encrypting our message. And that's what we're going to time here.

So the server-- you can think of this is Apache with open SSL-- you're going to get a message from the client, and you can think of this as a ciphertext c, or a

hypothetical ciphertext, that the client might have produced. And the first thing we're going to do with a ciphertext c, we want to decrypt it using roughly this formula. And if you remember the first optimization we're going to apply is the Chinese Remainder Theorem.

So the first thing we're going to do is basically split our pipeline in two parts. We're going to do one thing mod p another thing mod q and then recombine the results at the end of the day. So the first thing we're going to do is, we're actually going to take c and we're going to compute, let's call this c0, which is going to be equal to c mod q. And we're also going to have a different value, let's call it c1, which is going to be c mod p. And then we're going to do the same thing to each of these values to basically compute c to the d mod p and c to the d mod q.

And here we're going to basically initially we're going to [? starch. ?] After CRT, we're going to switch into Montgomery representation because that's going to make our multiplies very fast. So the next thing SSL is going to do to your number, it's actually going to compute all the [INAUDIBLE] at c0 prime, which is going to be c0 times R mod q.

And the same thing down here, I'm not going to write out the pipeline because that'll look the same. And then, now that we've switched into Montgomery form, we can finally do our multiplications. And here's where we're going to use the sliding window technique. So once we have c prime, we can actually try to compute this prime exponentiate it to 2d mod q. And here, as we're computing this value to the d, we're going to be using sliding windows. So here, we're going to do sliding windows for the bits in this d exponent.

And also we're going to do Karatsuba or regular multiplication depending on exactly what the size of our operands are. So if it turns out that the thing we're multiplying, c0 prime and maybe that previously squared result, are the same size, we're going to do Karatsuba. If c0 prime is tiny but some previous thing we're multiplying it to is big , then we're going to do quadratic multiplication, normal multiplication. There's sliding windows coming in here, here we also have this Karatsuba versus normal

multiplying.

And also in this step, the extra reductions come in. Because at every multiply, the extra reductions are going to be proportional to the thing we're exponentiating mod q. [INAUDIBLE] just plug in the formula over here, the probability extra reductions is going to be proportional to this value of c0 prime mod q divided by 2R. So this is where the really timing sensitive bit is going to come in. And there are actually two effects here. There's this Karatsuba versus normal choice. And then there's the number of extra reductions you're going to be making.

So we'll see how we exploit this in a second, but now that you get this result for mod q, you're going to get a similar result mod p, you can finally recombine these guys from the top and the bottom and use CRT. And what you get out from CRT is actually-- sorry I guess we need a first convert it back down into non Montgomery form. So we're going to get first, we're going to get c0 prime to the d divided by R mod q.

And this thing, because c0 prime was c0 times R mod q, if we do this then we're going to get back out our value of c to the d mod q. And we get c to the d here, we're going to get to c to the d mod p on the bottom version of this pipeline. And we can use CRT to get the value of c to the d mod m. Sorry for the small type here, or font size. But roughly it's the same thing we're expecting here. We can finally get our result. And we get our message, m.

So the server takes an incoming packet that it gets, runs it through this whole pipeline, does two parts of this pipeline, ends up with a decrypted message m that's equal c to the d mod m. And then it's going to check the padding of this message. And in this particular attack, because we're going to carefully construct this value c, the padding is going to actually not match up. We're going to choose the value c according to some other heuristics that aren't encrypting a real message with the correct padding.

So the padding is going to be a mismatch, and the server's going to need it to record an error back to the client. [? And it pulls ?] the connection. And that's the

time that we're going to measure to figure out how long this whole pipeline took. Makes sense? Questions about this pipeline and putting all the optimizations together?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, you're probably right. Yes, c1 to the d, c0 to the d. Yeah, this is c0. Yeah, correct.

AUDIENCE: When you divide by r [INAUDIBLE], isn't there a [INAUDIBLE] on how many q's you have to have to get the [? little bit ?] to be 0? [INAUDIBLE].

PROFESSOR: Yeah, so there might be extra reductions in this final phase as well. You're right. So potentially, we have do this divide by R correctly. So we probably have to do exactly the same thing as we saw for the Montgomery reductions here. When we do this divide by R to convert it back. So it's not clear exactly how many qs we should add. We should figure out how many qs to add, add that many, kill the low zeros, and then do mod q again, maybe an extra reduction. You're absolutely right, this is exactly the same kind of divide by R mod q as we do for every Montgomery multiplication step. Make sense? Any other questions?

All right. So how do you exploit this? How does an attacker actually figure out what the secret key of the server is by measuring the time of this entire pipeline? So these guys have a plan that basically involves guessing one bit of the private key at a time. And what they mean actually by guessing the private key is that you might think the private key is this encryption exponent d, because actually you know e, you know n, that's the public key. The only think you don't know is d. But in fact, in this attack they don't go for the exponent d directly, that's a little bit harder to guess.

Instead, what they're going to go for is the value q or the value p, doesn't really matter which one. Once you guess what the value p or q is, then you can give an n, you can factor in the p times q. Then if you know p times q, you can actually-- sorry-- if you know the values of p and q, you can compute that phi function we saw before. That's going to allow you to get the value d from the value e. So this

factorization of the value m is hugely important, it should be secret for RSA to remain secure. So these guys are actually going to go and try to guess what the value of q is by timing this pipeline. All right.

So how do these guys actually do it? Well, they construct carefully chosen inputs, c, into this pipeline and-- I guess I keep saying they keep measuring the time for this guy. But the particular, well, there's two parts of the attack, you have to bootstrap it a little bit to guess the first couple of bits. And then once you have the first couple of bits, you can I guess the next bit. So let me not say exactly how they guess the first couple of bits because it's actually much more interesting to see how they guess the next bit. And then we'll come back if we have time to look at how they guess the first couple of bits [? at this ?] in the paper.

But basically, suppose you have a guess g about what the bits are of this value q. So you know that q has some bits, g0, g1, g2, et cetera. And actually, I guess these are not even gs, these are real q bits, so let me write it as that. So you know tat q bit 0 q bit 1, q bit 2, these are the highest bits of q. And then you're trying to guess lower and lower bits. So suppose you know the value of q up to bit j. And from that point on, your guess is actually all 0. You have no idea what the other bits are.

So these guys are going to try to get this guess g into this place in the pipeline. Because this is where there are two tiny effects: this choice of Karatsuba versus normal multiplication. And this choice of, or this a different number of extra reductions depending on the value c0 prime. Sp they're going to actually try to get two different guess values into that place in the pipeline. One that looks like this, and one that they call g high, which is all the same high bits, q2 qj. And for the next bit, which they don't know, [? you ?] guess g is going to have 0, g high is going to have a bit 1 here and all zeros later on.

So how does it help these guys figure out what's going on? So there are really two ways you can think of it. Suppose that we get this guess g to be the value of c0 prime. We can think of g and g high being the c0 prime value on that left board over there. It's actually fairly straightforward to do this because c0 prime is pretty

deterministically computed from the input ciphertext c0. You just multiply it by R. So, in order for them to get some value to here, as a guess, they just need to take their guess and first divide it by R, so divide it by 2 to the 512 mod something. And then, they're going to inject it back. And the server's going to multiply it by R, and then off you go. Make sense? All right.

So suppose that we manage to get our particular chosen integer value into that c0 you're prime spot. So what's going to be the time to compute c0 prime to the d mod q? So there are two possible options here where q falls in this picture. So it might be that q is between these two values. Because the next bit of q is 0. So this value is going to be less than q, but this guy's going to be greater than q. So this happens if the next bit of q0 or it might be that q lies above both of these values if the next bit of q is 1. So now we can tell, OK, what's going to be the timing of decrypting these two values, if q lies in between them, or if q lies above both of them.

Let's look at the situation where q lies above both of them. Well in that case, actually everything is pretty much the same. Right? Because both of these values are smaller than q, then the value of these things mod q is going to be roughly the same. They're going to be a little bit different because this extra bit, but more or less they're the same magnitude.

And the number of extractor reductions is also probably not going to be hugely different because it's proportional to the value of this guy mod q. And for both these guys, they're both a little bit smaller than q, so they're all about the same. Neither of them is going to exceed q and all of a sudden have [? many or fewer ?] extra reductions.

So if q is greater than both of these guesses then Karatsuba versus normal is going to stay the same. The server is going to do the same thing basically for both g and g high in terms of Karatsuba versus normal. And the server's going to do about the same number of extra reductions for both these guys as well. So If you see that the server's taking the same amount of time to respond to these guesses, then you should probably guess that, oh, q probably has the bit 1 here.

On the other hand, if q lies in the middle, then there are two possible things that could trigger a change in the timing. One possibility is that because g high is just a little bit larger than q, then the number of extra reductions is going to be proportional to this guy mod q, which is very small because c0 prime is q plus just a little bit in these extra bits. So the number of extra reductions is going to [? flaunt it ?]. And all of a sudden, it will be faster.

Another possible thing that can happen is that maybe the server will decide, oh, now it's time to do normal multiplication instead of Karatsuba. Maybe for this value, all these, c to the 0 prime was the same number of bits as q if it turns out that g high is above q, then g high mod q is potentially going to have fewer bits. And if this crosses the [INAUDIBLE] boundary, then the server's going to do normal multiplication all of a sudden. So that's going to be in the other direction. So if you cross over, then normal multiplication kicks in, and things get a lot slower because normal multiplication is quadratic instead of nicer, faster Karatsuba. Question.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, because the number of extra reductions is proportional to from above there to c0 prime mod q. So if c0 prime, which is this value, is just a little over q. Then, this is tiny, as opposed to this guy who's basically the same as q, or all the high bits are the same as q, and then it's big. So then it'll be the difference that you can try to measure. So this is one interesting thing, actually a couple interesting things, these effects actually work in different directions, right. So if you hit a 32 bit boundary and Karatsuba versus normal switches, then all of a sudden it takes much longer to decrypt this message.

On the other hand, if it's not a 32 bit boundary, maybe this effect will tell you what's going on. So you actually have to watch for different effects. If you're not guessing a bit that's a multiple of 32 bits, then you should probably expect the time to drop because of extra reductions. On the other hand, if you're trying to guess a bit that's a multiple of 32, then maybe you should be expecting for it to jump a lot or maybe drop if it's [INAUDIBLE] normal.

So I guess what these guys look at in the paper, this actually doesn't really matter whether there's a jump up or a jump down in time. You should just expect if q is, if the next bit of q is 1, you should expect these things to take almost the same amount of time. And if the next bit of q is 0, then you should expect these guys to have a noticeable difference even if it's big or small, even if it's positive or negative.

So actually, they measure this. And it turns out to actually work pretty well. They have to do actually two interesting tricks to make this work out. If you remember the timing difference was tiny, it's an order of 1 to 2 microseconds. So it's going to be hard to measure this over a network, over an ethernet switch for example.

What they do is they actually do two kinds of measurements, two kinds of averaging. So for each guess that they send, they actually send it several times. In the paper, they said they send it like 7 times or something. So what kind of noise do you think this helps them with [? if they ?] just resend the same guess over and over? Yeah.

AUDIENCE: What's up with the [INAUDIBLE]?

PROFESSOR: Yeah, so if the network keeps adding different things, you just try the same thing many times. The thing in the server should be taking exactly the same amount of time every time and just average out the network noise. In the paper, they say they take the median value-- I actually don't understand why they take the median, I think they should be taking the min of the real thing that's going on-- but anyway, this was the average of the network.

But then they do this other weird thing, which is that when they're sending a guess, they don't just send the same guess 7 times, they actually send a neighborhood of guesses. And each value in the neighborhood gets sent 7 times itself. So they actually send g 7 times. Then they send g plus 1 also 7 times. Then they send g plus 2 also 7 times, et cetera, up to g plus 400 in the paper. Why do they do this kind of averaging as well over different g value instead of just sending g 7 times 400 times. Because it seems more straightforward. Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, that's actually what's going on. We're actually trying to measure exactly how long this piece of computation will take. But then there's lots of other stuff. For example, this other pipeline that's at the bottom is doing all the stuff mod p. I mean it's also going to take different amount of time depending on what exactly the input is.

So the cool thing is that if you perturb the value of all your guess g by adding 1, 2, 3, whatever, it's just [INAUDIBLE] the little bits. So the timing attack we just looked at just now, isn't going to change because that depended on this middle bit flipping.

But everything that's happening on the bottom side of the pipeline mod p is going to be totally randomized by this because when they do it mod p then adding an extra bit could shift things around quite a bit mod p. Then you're going to, it will average out other kinds of computational noise that's deterministic for a particular value but it's not related to this part of the computation we're trying to go after. Make sense?

AUDIENCE: How do they do that when they try to guess the lower bits?

PROFESSOR: So actually they use some other mathematical trick to only actually bother guessing the top half of the bits of q. It turns out if you know the top half of the bits of q there's some math you can rely on to factor the numbers, and then you're in good shape. So you can always [INAUDIBLE] little bit. Basically not worry about it. Make sense? Yeah, question.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, you're going to construct this value c0-- well you want the c0 prime-- you're going to construct a value c by basically taking your c0 prime and multiplying it times R inverse mod n. And then when the server takes this value, it's going to push it through here. So it's going to compute c0. It's going to be c mod q, so that value is going to be c0 prime R inverse mod q.

Then you multiply it by R, so you get rid of the R inverse. And then you end up with

a guess exactly in this position. So the cool thing is basically all manipulations leading up to here are just multiplying by this R. And you know what R is going be, it's going to be 2 to the 512. I'm going to be really straightforward. Make sense? Another question?

AUDIENCE: Could we just cancel out timing [INAUDIBLE]?

PROFESSOR: Well, if you do p, you'd be in business. Yeah, so that's the thing. Yeah, you don't know what p is, but you just want to randomize it out. Any questions? All right. [INAUDIBLE] but thanks for sticking around. So we'll start talking about other kinds of problems next week.