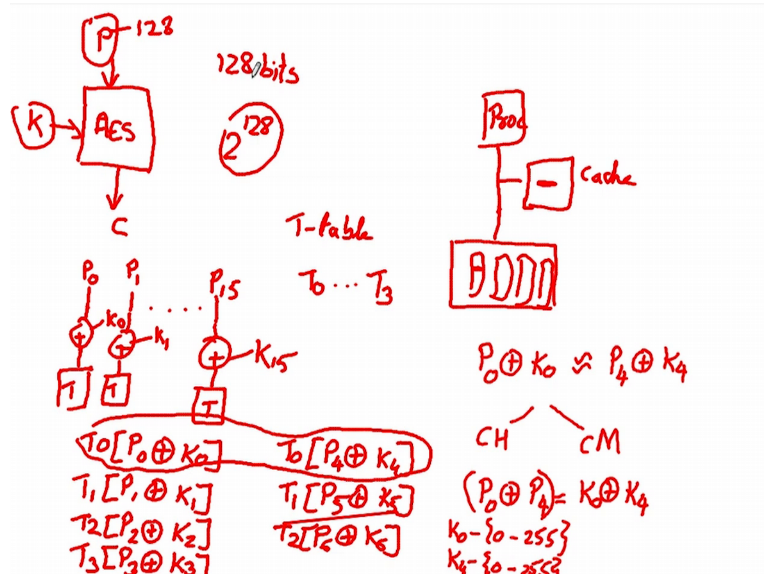
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Demo- Cache timing attack on T-table implementation of AES

Hello and welcome to this demonstration in the course for secure systems engineering. In the previous demonstration what we have seen was we had use the shared cache memory between 2 processes for communication we had shown how a sender and a receiver could actually exchange messages through the shared cache memory. Essentially the load installed operation to specify memory location could cause conflicts in the cache memory resulting in a variation in the execution time.

This variation in execution time was used to actually transfer information from one process to another. In this video we will take it one step further, we will show how cache memories can be also used to break cryptographic ciphers, we will demonstrate on a cipher like AES about how a few bits of the secret key can be recovered from the ciphers that is making it possible for an attack, so before we go into the attack we will actually just take a brief introduction on AES and we will see what exactly happens, so the introduction is just about sufficient to understand this specific attack, so AES as we know is a symmetric key cipher it is one of the most popular symmetric key ciphers and it is applied in varieties of different application.

(Refer Slide Time: 1:52)



The AES actually looks something like this… as a block diagram as a block it takes an input which is a plain text and it gives you a cipher text C. Now the plain text essentially is operated on by this AES operation, there is a secret key that is also involved and therefore the

cipher text which is obtained as the output of AES is a function of this plain text over here and the secret key. Now as we have seen in the other video lectures an attacker may actually know the plain text with a corresponding cipher text and he may also know the implementation which is used in AES. What is kept secret is this key, now a typical AES algorithm has a key cipher of 128 bits which means that there are 2 power 128 possible key ideas.

Breaking such a strong cipher would require several centuries of computing power constrained the amount of computing power that is available these days to actually break such a cipher. What people now show is that if an attacker is able to actually choose plain text or monitor the plain text that are being encrypted by AES and also monitor the execution time of AES then this huge pace of 2 power 128 can be reduced quite drastically. What the attacker would leverage is the timing behaviour of this AES implementation.

Now to understand what would happen we have to take a look at the AES implementation and a little more in detail about the AES algorithm, so 1st of all the plain text which is also of 128 bits is split into 16 bytes P naught to P15, once these bytes go into the cipher or during the implementation is that these bytes get XOR with 16 bytes of the secret key, so we will have like K 15 over here which is the 15th byte of the secret key and similarly we have K1 K naught and so on.

Now the next operation in the cipher is a lookup table, in the next operation in this implementation is memory access to a specific lookup table, this lookup table is known as the T table and essentially is defined as an integer array of 256 elements and would have the size of 1024 bytes, so we have a table over here like this and so on. This is about all that require to understand this cache timing attack, this specific implementation that we will use in this demonstration actually uses 5 T tables T0 to T4 out of these 5 tables the 1st 4 are important to us so these 4 tables are T0 to T3 the first accesses to the tables is as follows, so we have like 16 bytes P0 to P15 and 16 bytes of key which are XOR with their respective plain text bytes and then there are table accesses which are done as follows.

So we have T0 which gets XOR with P0 XOR with K0 then we have similarly T1 which is P1 XOR with K1 so what is happening over here is that we have this XOR operations which we have shown in the diagram and then there is based on the result of this XOR there is a lookup in this table at an index specified by P naught XOR K naught. Similarly there is the

table T2 which gets acted upon which gets looked up at the location P2 XOR K2 and T3 which is accessed at the location P3 XOR K3.

Now important for us are the next 4 accesses to the tables by the plain text P4 to P7 so these are done at locations T0 P4 XOR with K4, T1 gets accesses P5 XOR with K5. T2 accesses P6 XOR K6 and so on, so let us look at these tables look ups from a cache perspective, so initially let us just focus on this T0 table accesses that is these 2 table accesses. So initially we have this processor and as we know that there is a cache memory which caches some of the recently used data and then we have the large lower cache memory of the data.

So this is the cache memory, so these tables T0 to T3 are at the start of execution available in the DRAM, so now consider the axis to this tables at locations P naught XOR K naught and P4 XOR K4, so during the 1st access based on this index T0 XOR K0 some part of this table is loaded into the cache memory. Now during the 2nd access to the same table at the index P4 XOR K4 there are 2 things that can occur. Either you can get a cache hit which would mean that this index T4 XOR K4 is in the vicinity or is closed to the index P0 XOR K0, in such a case the processor would actually find the relevant information corresponding to those parts of the table and therefore would not require to actually go to the lower memories like that DRAM or the lower level cache memory.

So what we are saying is that if we just want to write it in a more mathematical sense we say that if P0 XOR K0 is approximately equal to P4 XOR K4 then we get a cache hit stop on the other hand if P0 XOR K0 is not equal to P4 XOR K4 then we get a cache miss, so notice that we have used the word approximately and given it this approximate signal because of the reason that P0 XOR K0 may fall in the same cache line as P4 XOR K4 in which case a cache hit could actually happen, so thus the 2nd operation P4 XOR K4 could result either in a cache hit or a cache miss, so what we have is this particular lookups either has a cache hit or a cache miss.

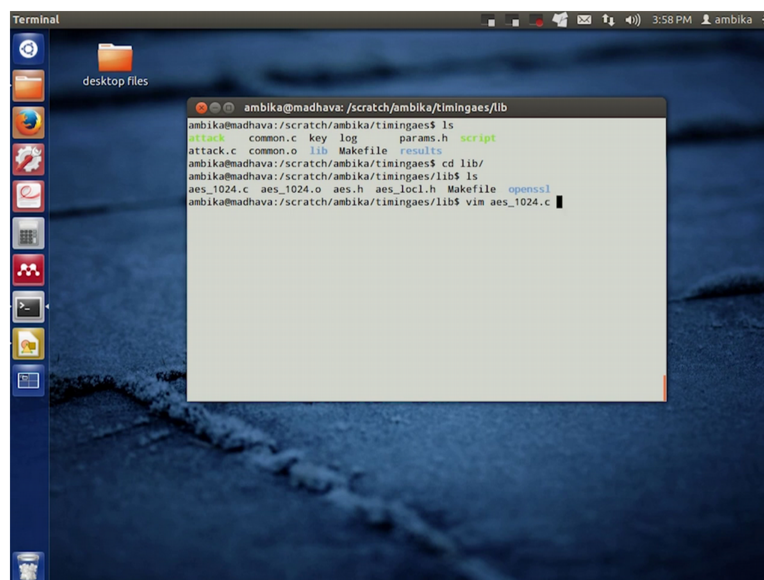
Now let us assume that there is a cache hit and for simplicity (())(10:23) this approximation with an equality thus we have P0 XOR K0 is equal to P4 XOR K4. Now if we just rearrange the terms we have like P0 XOR would be 4 to be equal to K0 XOR with K4. Now if the attacker actually knows the plain text bytes P0 and P4 it would indicate that he knows the XOR of the key bits K0 XOR K4. What this means is that the attacker has gained some information about secret key if you look at this in other way K0 we know is a byte therefore it has like 256 possible values from 0 to 255 and K4 is another independent byte which has

also 0 to 255 possible values, so without actually running this implementation if the attacker has to guess the values of K0 and K4 there are 512 different options.

On the other hand if the attacker runs the implementation and measures the execution time and identify cache hits and cache misses is uncertainty reduces from 512 to less than 256 without running the implementation if the attacker has to guess about K0 and K4 you see that there are 2 power 16 possibilities 2 power 8 for K0 and another 2 power 8 for K4 so together there would be like 2 power 16 possibilities, on the other hand if the attacker actually runs this implementation and is able to distinguish between a cache hit and cache miss or say by the timing channels that are present and then this reduces from 2 power 16 to 2 power 8.

Essentially what would be required is that the attacker guesses a value of K0 and then for that particular guess he can then compute the value K4 given this particular equation, so this is essentially the idea of this attack. Now this attack works very well with the older systems like the Intel, Core 2 Duo and so on but with modern systems like the i7 like we are going to have here as well as the i3 and i5 processors the attacks are not very successful. Nevertheless it can still be used to reduce the uncertainty about the key from 128 bits to something much more lower.

(Refer Slide Time: 13:09)



Terminal   3:59 PM ambika

desktop files

ambika@madhava: /scratch/ambika/timingaes/lib

```
Te2[x] = S [x].[01, 03, 02, 01];
Te3[x] = S [x].[01, 01, 03, 02];
Te4[x] = S [x].[01, 01, 01, 01];

Td0[x] = Si[x].[0e, 09, 0d, 0b];
Td1[x] = Si[x].[0b, 0e, 09, 0d];
Td2[x] = Si[x].[0d, 0b, 0e, 09];
Td3[x] = Si[x].[09, 0d, 0b, 0e];
Td4[x] = Si[x].[01, 01, 01, 01];
*/

static const u32 Te0[256] __attribute__((aligned(0x1000)))= {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
    0x8fcaca45U, 0x1f82829dU, 0x89c9c940U, 0xfa7d7d87U,
    0xeffafa15U, 0xb25959ebU, 0x8e4747c9U, 0xfbf0f00bU,
    0x41adadecU, 0xb3[illegible]d467U, 0x5fa2a2fdU, 0x45afafeaU,
    0x239c9cbfU, 0x53a4a4f7U, 0xe4727296U, 0x9bc0c05bU,
    0x75b7b7c2U, 0xe1fdfd1cU, 0x3d9393aeU, 0x4c26266aU,
    0x6c36365aU, 0x7e3f3f41U, 0xf5f7f702U, 0x83cccc4fU,
    0x6834345cU, 0x51a5a5f4U, 0xd1e5e534U, 0xf9f1f108U,
    0xe2717193U, 0xabd8d873U, 0x62313153U, 0x2a15153fU,
    0x0804040cU, 0x95c7c752U, 0x46232365U, 0x9dc3c35eU,
    0x30181828U, 0x379696a1U, 0x0a05050fU, 0x2f9a9ab5U,
    0x0e070709U, 0x24121236U, 0x1b80809bU, 0xdfe2e23dU,
    0xcdebeb26U, 0x4e272769U, 0x7fb2b2cdU, 0xea75759fU,
    0x1209091bU, 0x1d83839eU, 0x582c2c74U, 0x341a1a2eU,
    0x361b1b2dU, 0xdc6e6eb2U, 0xb45a5aeeU, 0x5ba0a0fbU,
    0xa45252f6U, 0x763b3b4dU, 0xb7d6d661U, 0x7db3b3ceU,
    0x5229297bU, 0xdde3e33eU, 0x5e2f2f71U, 0x13848497U,
    0xa65353f5U, 0xb9d1d168U, 0x00000000U, 0xc1eded2cU,
    0x40202060U, 0xe3fcfc1fU, 0x79b1b1c8U, 0xb65b5bedU,
    0xd46a6abeU, 0x8dcbcb46U, 0x67bebed9U, 0x7239394bU,
                                              58,18          3%
```

Terminal   3:59 PM ambika

desktop files

ambika@madhava: /scratch/ambika/timingaes/lib

```
    0x6fbabad5U, 0xf0787888U, 0x4a25256fU, 0x5c2e2e72U,
    0x381c1c24U, 0x57a6a6f1U, 0x73b4b4c7U, 0x97c6c651U,
    0xcbe8e823U, 0xa1dddd7cU, 0xe874749cU, 0x3e1f1f21U,
    0x964b4bddU, 0x61bdbddcU, 0x0d8b8b86U, 0x0f8a8a85U,
    0xe0707090U, 0x7c3e3e42U, 0x71b5b5c4U, 0xcc6666aaU,
    0x904848d8U, 0x06030305U, 0xf7f6f601U, 0x1c0e0e12U,
    0xc26161a3U, 0x6a35355fU, 0xae5757f9U, 0x69b9b9d0U,
    0x17868691U, 0x99c1c158U, 0x3a1d1d27U, 0x279e9eb9U,
    0xd9e1e138U, 0xebf8f813U, 0x2b9898b3U, 0x22111133U,
    0xd26969bbU, 0xa9d9d970U, 0x078e8e89U, 0x339494a7U,
    0x2d9b9bb6U, 0x3c1e1e22U, 0x15878792U, 0xc9e9e920U,
    0x87cece49U, 0xaa5555ffU, 0x50282878U, 0xa5dfdf7aU,
    0x038c8c8fU, 0x59a1a1f8U, 0x09898980U, 0x1a0d0d17U,
    0x65bfbfdaU, 0xd7e6e631U, 0x844242c6U, 0xd06868b8U,
    0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,
    0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU,
};
static const u32 Te1[256] __attribute__((aligned(0x1000))) = {
    0xa5c66363U, 0x84[illegible]87c7cU, 0x99ee7777U, 0x8df67b7bU,
    0x0dfff2f2U, 0xbdd66b6bU, 0xb1de6f6fU, 0x5491c5c5U,
    0x50603030U, 0x03020101U, 0xa9ce6767U, 0x7d562b2bU,
    0x19e7fefeU, 0x62b5d7d7U, 0xe64dababU, 0x9aec7676U,
    0x458fcacaU, 0x9d1f8282U, 0x4089c9c9U, 0x87fa7d7dU,
    0x15effafaU, 0xebb25959U, 0xc98e4747U, 0x0bfbf0f0U,
    0xec41adadU, 0x67b3d4d4U, 0xfd5fa2a2U, 0xea45afafU,
    0xbf239c9cU, 0xf753a4a4U, 0x96e47272U, 0x5b9bc0c0U,
    0xc275b7b7U, 0x1ce1fdfdU, 0xae3d9393U, 0x6a4c2626U,
    0x5a6c3636U, 0x417e3f3fU, 0x02f5f7f7U, 0x4f83ccccU,
    0x5c683434U, 0xf451a5a5U, 0x34d1e5e5U, 0x08f9f1f1U,
    0x93e27171U, 0x73abd8d8U, 0x53623131U, 0x3f2a1515U,
    0x0c080404U, 0x5295c7c7U, 0x65462323U, 0x5e9dc3c3U,
    0x28301818U, 0xa1379696U, 0x0f0a0505U, 0xb52f9a9aU,
    0x090e0707U, 0x36241212U, 0x9b1b8080U, 0x3ddfe2e2U,
    0x26cdebebU, 0x694e2727U, 0xcd7fb2b2U, 0x9fea7575U,
    0x1b120909U, 0x9e1d8383U, 0x74582c2cU, 0x2e341a1aU,
                                              107,18         8%
```

Terminal   3:59 PM ambika

desktop files

ambika@madhava: /scratch/ambika/timingaes/lib

```
    0xd56fbabaU, 0x88f07878U, 0x6f4a2525U, 0x725c2e2eU,
    0x24381c1cU, 0xf157a6a6U, 0xc773b4b4U, 0x5197c6c6U,
    0x23cbe8e8U, 0x7ca1ddddU, 0x9ce87474U, 0x213e1f1fU,
    0xdd964b4bU, 0xdc61bdbdU, 0x860d8b8bU, 0x850f8a8aU,
    0x90e07070U, 0x427c3e3eU, 0xc471b5b5U, 0xaacc6666U,
    0xd8904848U, 0x05060303U, 0x01f7f6f6U, 0x121c0e0eU,
    0xa3c26161U, 0x5f6a3535U, 0xf9ae5757U, 0xd069b9b9U,
    0x91178686U, 0x5899c1c1U, 0x273a1d1dU, 0xb9279e9eU,
    0x38d9e1e1U, 0x13ebf8f8U, 0xb32b9898U, 0x33221111U,
    0xbbd26969U, 0x70a9d9d9U, 0x89078e8eU, 0xa7339494U,
    0xb62d9b9bU, 0x223c1e1eU, 0x92158787U, 0x20c9e9e9U,
    0x4987ceceU, 0xffaa5555U, 0x78502828U, 0x7aa5dfdfU,
    0x8f038c8cU, 0xf859a1a1U, 0x80098989U, 0x171a0d0dU,
    0xda65bfbfU, 0x31d7e6e6U, 0xc6844242U, 0xb8d06868U,
    0xc3824141U, 0xb0299999U, 0x775a2d2dU, 0x111e0f0fU,
    0xcb7bb0b0U, 0xfca85454U, 0xd66dbbbbU, 0x3a2c1616U,
};
static const u32 Te2[256] __attribute__((aligned(0x1000))) = {
    0x63a5c663U, 0x7c[illegible]f87cU, 0x7799ee77U, 0x7b8df67bU,
    0xf20dfff2U, 0x6bbdd66bU, 0x6fb1de6fU, 0xc55491c5U,
    0x30506030U, 0x01030201U, 0x67a9ce67U, 0x2b7d562bU,
    0xfe19e7feU, 0xd762b5d7U, 0xabe64dabU, 0x769aec76U,
    0xca458fcaU, 0x829d1f82U, 0xc94089c9U, 0x7d87fa7dU,
    0xfa15effaU, 0x59ebb259U, 0x47c98e47U, 0xf00bfbf0U,
    0xadec41adU, 0xd467b3d4U, 0xa2fd5fa2U, 0xafea45afU,
    0x9cbf239cU, 0xa4f753a4U, 0x7296e472U, 0xc05b9bc0U,
    0xb7c275b7U, 0xfd1ce1fdU, 0x93ae3d93U, 0x266a4c26U,
    0x365a6c36U, 0x3f417e3fU, 0xf702f5f7U, 0xcc4f83ccU,
    0x345c6834U, 0xa5f451a5U, 0xe534d1e5U, 0xf108f9f1U,
    0x7193e271U, 0xd873abd8U, 0x31536231U, 0x153f2a15U,
    0x040c0804U, 0xc75295c7U, 0x23654623U, 0xc35e9dc3U,
    0x18283018U, 0x96a13796U, 0x050f0a05U, 0x9ab52f9aU,
    0x07090e07U, 0x12362412U, 0x809b1b80U, 0xe23ddfe2U,
    0xeb26cdebU, 0x27694e27U, 0xb2cd7fb2U, 0x759fea75U,
    0x091b1209U, 0x839e1d83U, 0x2c74582cU, 0x1a2e341aU,
                                              173,18        13%
```

```
ambika@madhava: /scratch/ambika/timingaes/lib
    0xbad56fbaU, 0x7888f078U, 0x256f4a25U, 0x2e725c2eU,
    0x1c24381cU, 0xa6f157a6U, 0xb4c773b4U, 0xc65197c6U,
    0xe823cbe8U, 0xdd7ca1ddU, 0x749ce874U, 0x1f213e1fU,
    0x4bdd964bU, 0xbddc61bdU, 0x8b860d8bU, 0x8a850f8aU,
    0x7090e070U, 0x3e427c3eU, 0xb5c471b5U, 0x66aacc66U,
    0x48d89048U, 0x03050603U, 0xf601f7f6U, 0x0e121c0eU,
    0x61a3c261U, 0x355f6a35U, 0x57f9ae57U, 0xb9d069b9U,
    0x86911786U, 0xc15899c1U, 0x1d273a1dU, 0x9eb9279eU,
    0xe138d9e1U, 0xf813ebf8U, 0x98b32b98U, 0x11332211U,
    0x69bbd269U, 0xd970a9d9U, 0x8e89078eU, 0x94a73394U,
    0x9bb62d9bU, 0x1e223c1eU, 0x87921587U, 0xe920c9e9U,
    0xce4987ceU, 0x55ffaa55U, 0x28785028U, 0xdf7aa5dfU,
    0x8c8f038cU, 0xa1f859a1U, 0x89800989U, 0x0d171a0dU,
    0xbfda65bfU, 0xe631d7e6U, 0x42c68442U, 0x68b8d068U,
    0x41c38241U, 0x99b02999U, 0x2d775a2dU, 0x0f111e0fU,
    0xb0cb7bb0U, 0x54fca854U, 0xbbd66dbbU, 0x163a2c16U,
};
static const u32 Te3[256] __attribute__((aligned(0x1000))) = {

    0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
    0xf2f20dffU, 0x6b6bbdd6U, 0x6f6fb1deU, 0xc5c55491U,
    0x30305060U, 0x01010302U, 0x6767a9ceU, 0x2b2b7d56U,
    0xfefe19e7U, 0xd7d762b5U, 0xababe64dU, 0x76769aecU,
    0xcaca458fU, 0x82829d1fU, 0xc9c94089U, 0x7d7d87faU,
    0xfafa15efU, 0x5959ebb2U, 0x4747c98eU, 0xf0f00bfbU,
    0xadadec41U, 0xd4d467b3U, 0xa2a2fd5fU, 0xafafea45U,
    0x9c9cbf23U, 0xa4a4f753U, 0x727296e4U, 0xc0c05b9bU,
    0xb7b7c275U, 0xfdfd1ce1U, 0x9393ae3dU, 0x26266a4cU,
    0x36365a6cU, 0x3f3f417eU, 0xf7f702f5U, 0xcccc4f83U,
    0x34345c68U, 0xa5a5f451U, 0xe5e534d1U, 0xf1f108f9U,
    0x717193e2U, 0xd8d873abU, 0x31315362U, 0x15153f2aU,
    0x04040c08U, 0xc7c75295U, 0x23236546U, 0xc3c35e9dU,
    0x18182830U, 0x9696a137U, 0x05050f0aU, 0x9a9ab52fU,
    0x0707090eU, 0x12123624U, 0x80809b1bU, 0xe2e23ddfU,
    0xebeb26cdU, 0x2727694eU, 0xb2b2cd7fU, 0x75759feaU,
                                              239,18        18%
```

```
ambika@madhava: /scratch/ambika/timingaes/lib
    0x4141c382U, 0x9999b029U, 0x2d2d775aU, 0x0f0f111eU,
    0xb0b0cb7bU, 0x5454fca8U, 0xbbbbd66dU, 0x16163a2cU,
};
static const u32 Te4[256] = {
    0x63636363U, 0x7c7c7c7cU, 0x77777777U, 0x7b7b7b7bU,
    0xf2f2f2f2U, 0x6b6b6b6bU, 0x6f6f6f6fU, 0xc5c5c5c5U,
    0x30303030U, 0x01010101U, 0x67676767U, 0x2b2b2b2bU,
    0xfefefefeU, 0xd7d7d7d7U, 0xababababU, 0x76767676U,
    0xcacacacaU, 0x82828282U, 0xc9c9c9c9U, 0x7d7d7d7dU,
    0xfafafafaU, 0x59595959U, 0x47474747U, 0xf0f0f0f0U,
    0xadadadadU, 0xd4d4d4d4U, 0xa2a2a2a2U, 0xafafafafU,
    0x9c9c9c9cU, 0xa4a4a4a4U, 0x72727272U, 0xc0c0c0c0U,
    0xb7b7b7b7U, 0xfdfdfdfdU, 0x93939393U, 0x26262626U,
    0x36363636U, 0x3f3f3f3fU, 0xf7f7f7f7U, 0xccccccccU,
    0x34343434U, 0xa5a5a5a5U, 0xe5e5e5e5U, 0xf1f1f1f1U,
    0x71717171U, 0xd8d8d8d8U, 0x31313131U, 0x15151515U,
    0x04040404U, 0xc7c7c7c7U, 0x23232323U, 0xc3c3c3c3U,
    0x18181818U, 0x96969696U, 0x05050505U, 0x9a9a9a9aU,
    0x07070707U, 0x12[illegible]21212U, 0x80808080U, 0xe2e2e2e2U,
    0xebebebebU, 0x27272727U, 0xb2b2b2b2U, 0x75757575U,
    0x09090909U, 0x83838383U, 0x2c2c2c2cU, 0x1a1a1a1aU,
    0x1b1b1b1bU, 0x6e6e6e6eU, 0x5a5a5a5aU, 0xa0a0a0a0U,
    0x52525252U, 0x3b3b3b3bU, 0xd6d6d6d6U, 0xb3b3b3b3U,
    0x29292929U, 0xe3e3e3e3U, 0x2f2f2f2fU, 0x84848484U,
    0x53535353U, 0xd1d1d1d1U, 0x00000000U, 0xededededU,
    0x20202020U, 0xfcfcfcfcU, 0xb1b1b1b1U, 0x5b5b5b5bU,
    0x6a6a6a6aU, 0xcbcbcbcbU, 0xbebebebeU, 0x39393939U,
    0x4a4a4a4aU, 0x4c4c4c4cU, 0x58585858U, 0xcfcfcfcfU,
    0xd0d0d0d0U, 0xefefefefU, 0xaaaaaaaaU, 0xfbfbfbfbU,
    0x43434343U, 0x4d4d4d4dU, 0x33333333U, 0x85858585U,
    0x45454545U, 0xf9f9f9f9U, 0x02020202U, 0x7f7f7f7fU,
    0x50505050U, 0x3c3c3c3cU, 0x9f9f9f9fU, 0xa8a8a8a8U,
    0x51515151U, 0xa3a3a3a3U, 0x40404040U, 0x8f8f8f8fU,
    0x92929292U, 0x9d9d9d9dU, 0x38383838U, 0xf5f5f5f5U,
    0xbcbcbcbcU, 0xb6b6b6b6U, 0xdadadadaU, 0x21212121U,
                                              320,18        24%
```

```
ambika@madhava: /scratch/ambika/timingaes/lib
        noaccess = 0;
        for(i=0; i<256; ++i)if(hc3[i] == 0) noaccess++;
        nmisses += 256 - noaccess;

        memset(hc0, 0, sizeof(hc0));
        memset(hc1, 0, sizeof(hc1));
        memset(hc2, 0, sizeof(hc2));
        memset(hc3, 0, sizeof(hc3));
        return nmisses;
}
#else
#define peek(a,b,c,d)          0
#define printnmisses()
#endif

/*
 * Encrypt a single block
 * in and out can overlap
 */
void AES_encrypt(const unsigned char *in, unsigned char *out,
                 const AES_KEY *key) {

        const u32 *rk;
        u32 s0, s1, s2, s3, t0, t1, t2, t3;
#ifndef FULL_UNROLL
        int r;
#endif /* ?FULL_UNROLL */

        assert(in && out && key);
        rk = (u32 *)key->rd_key;

        /*
         * map byte array block to cipher state
         * and add initial round key:
         */
        s0 = GETU32(in     ) ^ rk[0];
        s1 = GETU32(in +  4) ^ rk[1];
        s2 = GETU32(in +  8) ^ rk[2];
        s3 = GETU32(in + 12) ^ rk[3];
#ifdef FULL_UNROLL
                                              939,20-34     71%
```

So what we will see now is the actual code and a demonstration of the attack. This code may not work on the virtual machine that was given to you along with the course and you may to tweak up this program a little bit and in order to get it run on your (())(13:25). So we will be sharing this entire code with you, this code comprises of the attack.c which is essentially the attack and there is also a library that is present contains the AES implementation, so we will actually take a look at this AES implementation.

So this AES code is built on the lines of open SSL, one of the earlier limitations of the open SSL and the 1st thing to note is the tables T0 to T4 so they are defined here as follows, so each of these tables is of 32 bits and there are 256 entries in total, there are 1024 elements in this particular table. Similar to this table we have the 2nd table Te1 which is also a 1024 bytes, Te2 and Te3, so Te4 is also present but we will not be using this and the other tables I use for decryption which is not going to be important for us.

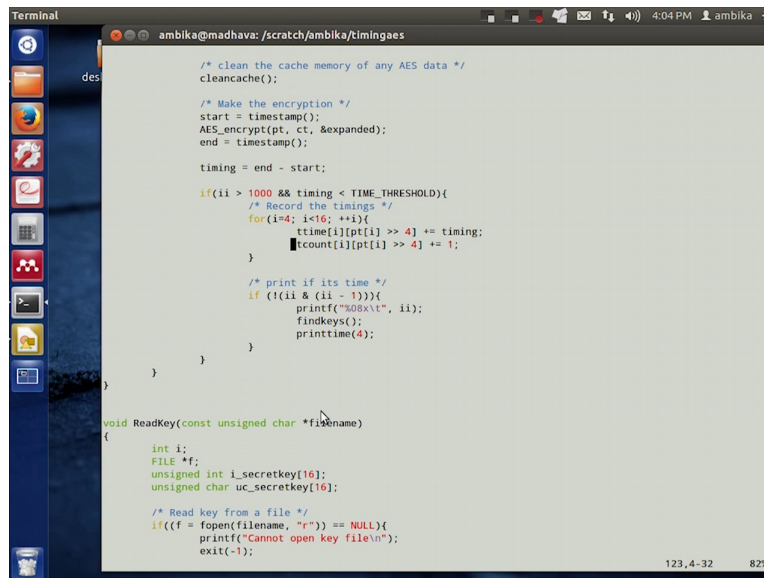
Let us go straight away to this AES encryption algorithm, so what this algorithm takes is an input which is the plain text. It takes the AES key this is a structured to the expanded key and it performs the encryption and find the use of cipher text through this particular output. The AES has several different rounds of operations but what is important for us is only these set of operations, these 4 lines essentially would XOR the secret key present in this AES key and a pointer is obtained gained over here with the inputs, so the inputs are here the 16 bytes of input and the (())(15:23) keys are XOR with it. During the 1st round operation as we had mentioned there are several table lookups, so in fact each of these 4 tables Te0, 1, 2 and 3 would have 4 lookups each, so the results are XOR and then passed on the other rounds of the

cipher, so the remaining part of cipher from this cache timing attack perspective is not very important for us.

(Refer Slide Time: 15:55)

```
ambika@madhava:/scratch/ambika/timingaes$

ambika@madhava:/scratch/ambika/timingaes$ ls
attack    common.c  key  log       params.h  script
attack.c  common.o  lib  Makefile  results
ambika@madhava:/scratch/ambika/timingaes$ cd lib/
ambika@madhava:/scratch/ambika/timingaes/lib$ ls
aes_1024.c  aes_1024.o  aes.h  aes_locl.h  Makefile  openssl
ambika@madhava:/scratch/ambika/timingaes/lib$ vim aes_1024.c
ambika@madhava:/scratch/ambika/timingaes/lib$ make
gcc  -O3 -I. aes_1024.c -c
ambika@madhava:/scratch/ambika/timingaes/lib$ ls
aes_1024.c  aes_1024.o  aes.h  aes_locl.h  Makefile  openssl
ambika@madhava:/scratch/ambika/timingaes/lib$ cd ..
ambika@madhava:/scratch/ambika/timingaes$ ls
attack  attack.c  common.c  common.o  key  lib  log  Makefile  params.h  results  script
ambika@madhava:/scratch/ambika/timingaes$ vim attack
```

```
	for (c=4; c<16; ++c)
		printf("%02d(%x) ", c, finddeviant(c));
	printf("\n");
}

double attackrnd1()
{
	int ii=0, i;
	unsigned int start, end, timing;

	while(ii++ <= (ITERATIONS)){
		/* Set a random plaintext */
		for(i=0; i<16; ++i) pt[i] = random() & 0xff;
		/* Fix a few plaintext bits of some plaintext bytes */
		pt[0] = pt[0] & 0x0f;
		pt[1] = pt[1] & 0x0f;
		pt[2] = pt[2] & 0x0f;
		pt[3] = pt[3] & 0x0f;

		/* clean the cache memory of any AES data */
		cleancache();

		/* Make the encryption */
		start = timestamp();
		AES_encrypt(pt, ct, &expanded);
		end = timestamp();

		timing = end - start;

		if(ii > 1000 && timing < TIME_THRESHOLD){
			/* Record the timings */
			for(i=4; i<16; ++i){
				ttime[i][pt[i] >> 4] += timing;
				tcount[i][pt[i] >> 4] += 1;
			}

			/* print if its time */
			if (!(ii & (ii - 1))){
				printf("%08x\t", ii);
				findkeys();
                                                  115,17-31      68%
```



So before we go further we would make this particular AES implementation by just running a make, so this would create an object file AES 1024.o which we then involved in our attack, so we will now go to the attack code it is over here and we will just jump directly to the attack function and what we see is the following, so 1st of all over here we selection some random plain text, this plain text PT is defined globally as in array of 16 bytes.

We randomly select something on it and for certain bytes the plain text 0, 1, 2 and 3 we make the higher (())(16:40) to be equal to 0 then we invoke this function called clean cash and then we invoke this AES encrypt. Now what happens during the AES encryption is there is this plain text which is taken as the 1st parameter and there is an expanded key which is also available and finally this would trigger the AES encryption to occur and the cipher text is obtained.

These times stamps here as well as here are used to actually time the execution of this AES, so you could refer to the previous video about the cover channels to look at how these timestamps are programmed and how they obtain the time. More important for us is that we evaluate these timings and use a similar frequency distribution and statistical techniques has been done previously to record the timing and then at a later point determine the secret key. We will not go more into detail about this particular program and it is actually quite interesting and you could look at it more in detail and try to run this particular program.

(Refer Slide Time: 17:59)

In order to compile this program we run a make and obtain an output known as attack. The idea is that we run this attack and try to get this security key, so the secret key is present in this particular file call key, this file is actually ready by the AES code and it is used during the encryption. The idea is to get as many bytes as possible from this secret key, the first think to note is that there 16 such bytes over here and therefore we would obtain a key size of 128 bits, the hope is that when we run this attack we would be able to reduce this uncertainty about the key to a significant level.

So in order to run we just run the attack and what gets printed are the potential key bytes from here onwards that is 4th to the 15th key bytes. The values printed within the brackets are what the attack program has identified the 4th key which is 64 for instance. The attack

program has identified 6 essentially has been able to identify the most significant 4 bits of the key.

Now each row over here tells the number of measurements that have taken place, so for example this is in hexadecimal notation, so for example this particular row are the results of the attack after 1024 iteration that is 1024 measurements of the execution time of the AES and what we see is that the next one is double that amount which is around 2048 iterations and so on, so as we increase the number of timing measurements that is the iterations in the attack actually increase, we see that the result become more and more accurate.

So this particular code is program to go up to 2 power 24 and these are the results after those titrations, so what we see is that we have this enable which has been predicted correctly by the attack this is 6 over here and what the attack also obtain is 6 however the next byte which is 50 the attack has obtain 7 wages, so in this particular way we can find the correctness of the attack, so we can stop this attack due to time constraints and see the number of bytes that have actually been obtained and count the number of bytes that have actually been obtained correctly, so for example here this bike has been correctly obtained because this is 6 so on this bytes has been found completely, so what we find if we compare the most significant label of these 16 bytes and compared them with the results within the braces over here, we find that the attack has identified 4 nibbles correctly that these nibbles occur here, here, here and finally here.

(Refer Slide Time: 21:33)

K 128 bits

4*4 = 16

112 bits

So let us see how much we have actually know about the key, so prior to the timing attack we had a key size of 128 bits which means that there are… to our uncertainty about the key is essentially 128 bits. The actual key is one among 2 power 128 possibilities, so after the timing attack which we have done we found that there are 4 nibbles that we have predicted correctly, so therefore it is like we have identified 4 that is 16 bits of the secret key, so thus the uncertainty about the key reduces from 128 bits to 128 bits minus 8, so this means the uncertainty about the secret key has reduced to 112 bits.

So thus we see that we have been able to reduce the uncertainty about the key, this is one of the initial attacks that was actually proposed for cache memory. There are much more advanced attack where you can get the secretly key with much more accuracy we leave it to the viewers to actually try this particular attack and also read the state-of-the-art attacks in this direction and see how we could reduce the entropy or the uncertainty about the secret key to construct lesser than what we obtain over here. Thank you.