Cache-timing attacks on AES

Daniel J. Bernstein [⋆]

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
djb@cr.yp.to

Abstract. This paper demonstrates complete AES key recovery from known-plaintext timings of a network server on another computer. This attack should be blamed on the AES design, not on the particular AES library used by the server; it is extremely difficult to write constant-time high-speed AES software for common general-purpose computers. This paper discusses several of the obstacles in detail.

Keywords: side channels, timing attacks, software timing attacks, cache timing, load timing, array lookups, S-boxes, AES

1 Introduction

This paper reports successful extraction of a complete AES key from a network server on another computer. The targeted server used its key solely to encrypt data using the OpenSSL AES implementation on a Pentium III.

The successful attack was a very simple timing attack. Presumably the same technique can extract complete AES keys from the more complicated servers actually used to handle Internet data, although the attacks will often require extra timings to average out the effects of variable network delays.

Are attacks of this type limited to the Pentium III? No. Every chip I have tested—an AMD Athlon, an Intel Pentium III, an Intel Pentium M, an IBM PowerPC RS64 IV, and a Sun UltraSPARC III—has shown comparable levels of OpenSSL AES timing variability. Presumably it is possible to carry out similar attacks against software running on all of these CPUs. I chose to focus on the Pentium III because the Pentium III is one of the most common CPUs in today's Internet servers.

Was there some careless mistake in the OpenSSL implementation of AES? No. The problem lies in AES itself: it is extremely difficult to write constant-time

[⋆] The author was supported by the National Science Foundation under grant CCR–9983950, and by the Alfred P. Sloan Foundation. Date of this document: 2005.04.14. Permanent ID of this document: `cd9faae9bd5308c440df50fc26a517b4`. This is a preliminary version meant to announce ideas; it will be replaced by a final version meant to record the ideas for posterity. There may be big changes before the final version. Future readers should not be forced to look at preliminary versions, unless they want to check historical credits; if you cite a preliminary version, please repeat all ideas that you are using from it, so that the reader can skip it.

high-speed AES software for common general-purpose CPUs. The underlying problem is that it is extremely difficult to load an array entry in time that does not depend on the entry's index. I chose to focus on OpenSSL because OpenSSL is one of today's most popular cryptographic toolkits.

(Constant time *low-speed* AES software is fairly easy to write but is also unacceptable for many applications. Rijndael would obviously not have survived the first round of AES competition if it had been implemented in this way.)

Is AES the only cryptographic function vulnerable to this type of attack? No. The relevant feature of AES software, namely its heavy reliance upon S-boxes, is shared by cryptographic software for many other functions. For example, [18, Section 3.2]—"GCM"—does not achieve tolerable speed without 4096-byte S-boxes; presumably this GCM software leaks a great deal of secret information to a simple timing attack.

But there are exceptions. Consider the Tiny Encryption Algorithm, published by Wheeler and Needham in [24]; SHA-256, published in [1]; Helix, published by Ferguson, Whiting, Schneier, Kelsey, Lucks, and Kohno in [11]; and my new Salsa20. These cryptographic functions are built from a few simple operations that take constant time on common general-purpose CPUs: 32-bit additions, constant-distance rotations, etc. There is no apparent incentive for implementors of these functions to use S-box lookups or other operations with input-dependent timings; top speed is easily achieved by constant-time software. This paper can be interpreted as a call for further research into such functions.

Section 2 of this paper reviews AES. Sections 3 through 6 present the attack in detail. Section 7 identifies the relevant errors in the AES standardization process. Sections 8 through 15 focus on the extremely difficult problem of writing constant-time high-speed AES software.

I make no claims of novelty for the basic observation that memory-access timings can leak secret information. But these timings are

- considerably more complicated,
- considerably easier to exploit, and
- much more difficult to control

than indicated in the previous literature on timing attacks. Using secret data as an array index is a recipe for disaster.

A note on terminology

In this paper, **S-box lookups** are table lookups using input-dependent indices; i.e., loads from input-dependent addresses in memory.

S-boxes are a feature of software, not of the mathematical functions computed by that software. For example, one can write AES software that does not use S-boxes, or (much faster) AES software that uses S-boxes.

Some cryptographers refer to various mathematical functions as "S-boxes." I have no idea how this competing notion of "S-box" is defined. Is the 16-bit-to-8-bit function $(x, y) \mapsto x \oplus y$ an "S-box"? Perhaps there is a coherent definition of what it means for a function to be an "S-box," and perhaps that definition is useful in some contexts, but it is clearly not the right concept for this paper.

2 Summary of AES

AES scrambles a 16-byte input n using a 16-byte key k, a constant 256-byte table $S = (99, 124, 119, 123, 242, \dots)$, and another constant 256-byte table $S' = (198, 248, 238, 246, 255, \dots)$. These two 256-byte tables are expanded into four 1024-byte tables T_0, T_1, T_2, T_3 defined by

$$\begin{aligned} T_0[b] &= (S'[b], S[b], S[b], S[b] \oplus S'[b]),\\ T_1[b] &= (S[b] \oplus S'[b], S'[b], S[b], S[b]),\\ T_2[b] &= (S[b], S[b] \oplus S'[b], S'[b], S[b]),\\ T_3[b] &= (S[b], S[b], S[b] \oplus S'[b], S'[b]). \end{aligned}$$

Here $\oplus$ means xor, i.e., addition of vectors modulo 2.

AES works with two 16-byte auxiliary arrays, x and y. The first array is initialized to k. The second array is initialized to $n \oplus k$.

AES first modifies x as follows. View x as four 4-byte arrays x_0, x_1, x_2, x_3. Compute the 4-byte array $e = (S[x_3[1]] \oplus 1, S[x_3[2]], S[x_3[3]], S[x_3[0]])$. Replace (x_0, x_1, x_2, x_3) with $(e \oplus x_0, e \oplus x_0 \oplus x_1, e \oplus x_0 \oplus x_1 \oplus x_2, e \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3)$.

AES then modifies y as follows. View y as four 4-byte arrays y_0, y_1, y_2, y_3. Replace (y_0, y_1, y_2, y_3) with

$$\begin{aligned} (&T_0[y_0[0]] \oplus T_1[y_1[1]] \oplus T_2[y_2[2]] \oplus T_3[y_3[3]] \oplus x_0,\\ &T_0[y_1[0]] \oplus T_1[y_2[1]] \oplus T_2[y_3[2]] \oplus T_3[y_0[3]] \oplus x_1,\\ &T_0[y_2[0]] \oplus T_1[y_3[1]] \oplus T_2[y_0[2]] \oplus T_3[y_1[3]] \oplus x_2,\\ &T_0[y_3[0]] \oplus T_1[y_0[1]] \oplus T_2[y_1[2]] \oplus T_3[y_2[3]] \oplus x_3). \end{aligned}$$

I learned this view of AES from software published by Barreto in [4].

AES modifies x again, using $\oplus 2$ instead of $\oplus 1$; modifies y again; modifies x again, using $\oplus 4$; modifies y again; and so on for a total of ten rounds. The constants for the x modifications are $1, 2, 4, 8, 16, 32, 64, 128, 27, 54$. On the tenth round, the y modification uses $(S[], S[], S[], S[])$ rather than $T_0[] \oplus T_1[] \oplus T_2[] \oplus T_3[]$. The final value of y is the output $\text{AES}_k(n)$.

Note that the evolution of x is independent of n. One can expand k into the eleven values of x, together occupying 176 bytes, and then reuse those values for each n. This key expansion is not always a good idea; if many keys are handled simultaneously then the time to load the precomputed values of x from memory may exceed the time needed to recompute those values.

3 Summary of the attack

Here is the simplest conceivable timing attack on AES.

Consider the variable-index array lookup $T_0[k[0] \oplus n[0]]$ near the beginning of the AES computation. One might speculate that the time for this array lookup depends on the array index; that the time for the whole AES computation is well correlated with the time for this array lookup; that, consequently, the AES timings leak information about $k[0] \oplus n[0]$; and that one can deduce the exact value of $k[0]$ from the distribution of AES timings as a function of $n[0]$. Similar comments apply to $k[1] \oplus n[1]$, $k[2] \oplus n[2]$, etc.

Assume, for example, that the attacker

- watches the time taken by the victim to handle many n's,
- totals the AES times for each possible $n[13]$, and
- observes that the overall AES time is maximum when $n[13]$ is, say, 147.

Assume that the attacker also observes, by carrying out experiments with known keys k on a computer with the same AES software and the same CPU, that the overall AES time is maximum when $k[13] \oplus n[13]$ is, say, 8. The attacker concludes that the victim's $k[13]$ is $147 \oplus 8 = 155$.

I tried this embarrassingly simple AES timing attack against a simple server and discovered that it worked. Sections 4 through 6 explain exactly what I did, so that others can reproduce my results.

4 The targeted server

The complete server software, `server.c`, is shown in Appendix A. The server requires an x86 computer running UNIX.

The server is given a 16-byte AES key and an IP address. It listens for UDP packets sent to port 10000 of that IP address. Packets have variable length but always begin with a 16-byte nonce. The server copies the nonce to its response, along with a high-precision receipt timestamp obtained from the CPU's cycle counter:

```
*(unsigned int *) (out + 32) = timestamp();
...
for (i = 0;i < 16;++i) out[i] = in[i];
```

The server copies the rest of the packet into a work area having `3*len` bytes, where `len` is the packet length:

```
unsigned char workarea[len * 3];
...
for (i = 16;i < len;++i) workarea[i] = in[i];
```

The server also scrambles the nonce using OpenSSL:

```
AES_encrypt(in,workarea,&expanded);
```

A real server would next use the scrambled nonce to verify an authenticator for the incoming packet, and would then perform some useful work for authenticated packets. My server simply sends back a scrambled zero (precomputed), along with an outgoing timestamp:

```
for (i = 0;i < 16;++i) out[16 + i] = scrambledzero[i];
*(unsigned int *) (out + 36) = timestamp();
```

The server does not send any other information to the client, and does not use its key in any other way.

Of course, I wrote this server to minimize the amount of noise in the timings available to the client. However, adding noise does not stop the attack: the client simply averages over a larger number of samples, as in [7]. In particular, reducing the precision of the server's timestamps, or eliminating them from the server's responses, does not stop the attack: the client simply uses round-trip timings based on its local clock, and compensates for the increased noise by averaging over a larger number of samples.

5 Preparation for the attack

In the attacker's role, I compiled and ran the server on an 850MHz Pentium III desktop computer running FreeBSD 4.8:

```
% gcc --version
2.95.4
% openssl version
OpenSSL 0.9.7a Feb 19 2003
% gcc -O3 -o server server.c -lcrypto
% printenv | wc -c
     549
% ./server 192.168.123.141 < /dev/zero
```

At this point the server was listening for UDP packets on port 1000 of IP address 192.168.123.141. It was using a known AES key, namely all zeros.

Then, from another computer, I sent random 400-byte packets to the server, using the `study` program (also x86-specific) shown in Appendix B:

```
% gcc -O3 -o study study.c -lm
% ./study 192.168.123.141 400 > study.400
```

The program printed many lines such as the following:

```
13  400   8 4196901 3087.895 129.512 3.108 0.063
```

This line conveys the following information: The server was sent 4196901 400-byte packets having $n[13] = 8$. It handled those packets in 3087.895 cycles on average, with a deviation of 129.512 cycles. Compared to the average over all choices of $n[13]$, the average for $n[13] = 8$ was 3.108 cycles higher. The number 0.063 is an estimate of the deviation in this difference: if the distribution of timings is close to normal, with a deviation around 129.512 cycles, then an average of 4196901 such timings has a deviation around $129.512/\sqrt{4196901} \approx 0.063$ cycles. (The deviation in the average over all choices of $n[13]$ is, presumably, 1/16th as large, and is ignored.)

Other lines show similar information for other choices of $n[13]$. Here are the top cycle counts: 3.108 above average for $n[13] = 8$; 2.763 above average for

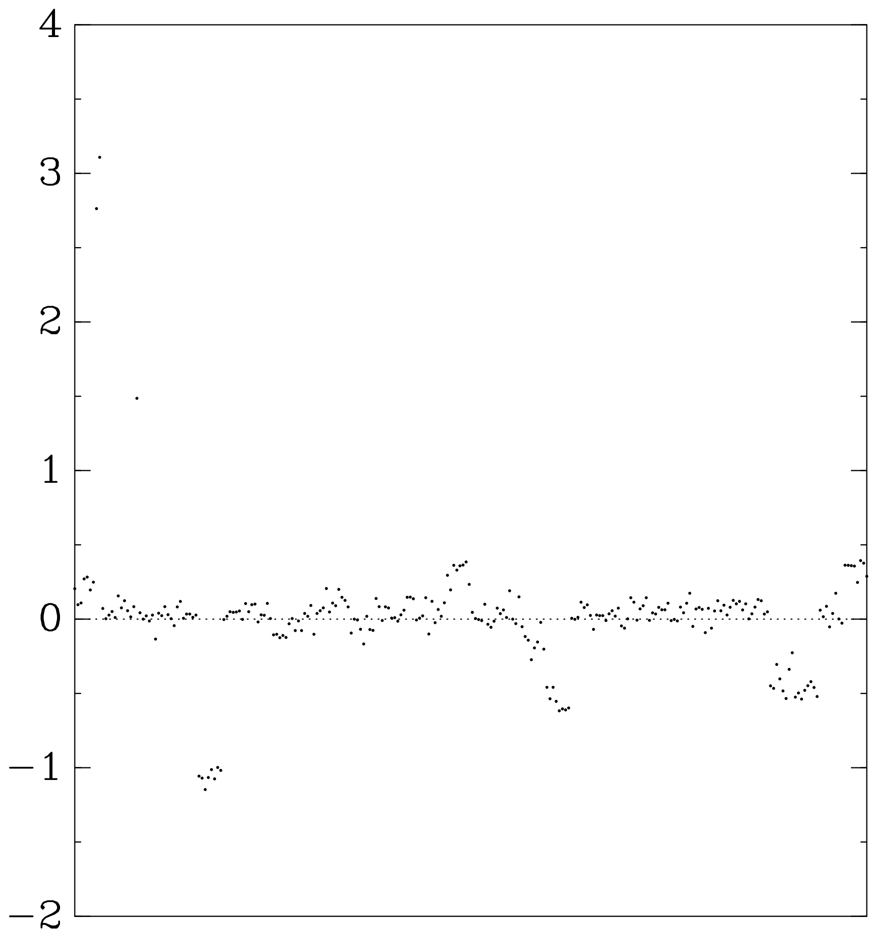

Fig. 5.1. How $n[13]$ affects OpenSSL AES timings for $k = 0$ on a Pentium III inside the targeted server. The horizontal axis is the choice of $n[13]$, from 0 through 255. The vertical axis is the average number of cycles for about 2^{22} 400-byte packets with that choice of $n[13]$, minus the average number of cycles for all choices of $n[13]$.

$n[13] = 7$; 1.486 above average for $n[13] = 20$; and 0.385 above average for $n[13] = 126$. Figure 5.1 shows these numbers for all choices of $n[13]$.

Presumably, for any key k, the graph of timings as a function of $n[13] \oplus k[13]$ looks the same as Figure 5.1. At this point I guessed that, for any key k, the choice of $n[13]$ that maximized the timings would be $k[13] \oplus 8$, immediately leaking the byte $k[13]$. This guess later turned out to be correct.

To analyze the entire pattern of timings rather than just the maximum, I computed, for each $i \in \{0, 1, \ldots, 255\}$, the correlation $\sum_{j \in \{0,1,\ldots,255\}} t(j)u(i \oplus j)$. Here $t(j)$ is the average of the first $\approx 2^{21}$ timings with $n[13] = j$, and $u(j)$ is the average of the next $\approx 2^{21}$ timings with $n[13] = j$. The correlation for $i = 0$ is about 36 squared cycles above average; the correlation for $i = 15$ is about 22 squared cycles above average; the correlations for $i \in \{1, 2, \ldots, 7\}$ are about 18 squared cycles above average; the correlations for other i's are never more than 11 squared cycles above average. For comparison, the estimated deviation

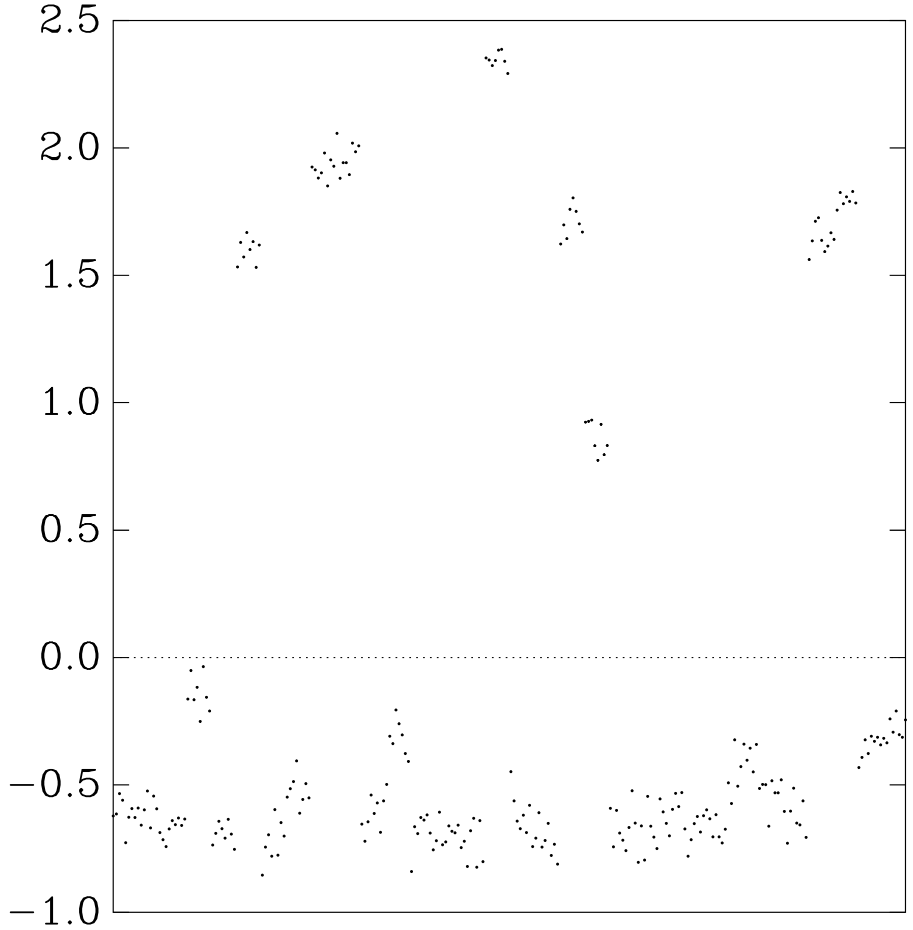

Fig. 5.2. How $n[5]$ affects OpenSSL AES timings for $k = 0$ on a Pentium III inside the targeted server. The horizontal axis is the choice of $n[5]$, from 0 through 255. The vertical axis is the average number of cycles for about 2^{22} 400-byte packets with that choice of $n[5]$, minus the average number of cycles for all choices of $n[5]$.

in these correlations is about 0.3 squared cycles. Thus the timings distinguish $n[13]$ from $n[13] \oplus i$ for all $i \neq 0$.

The output of the `study` program also covered $n[b]$ for $b \neq 13$. For example, Figure 5.2 shows the time differences for $b = 5$. In this case, the correlations for $i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ are about 268 squared cycles, and are close enough together to be statistically indistinguishable. The correlations for other i's are much smaller, never more than 150 squared cycles; in other words, the timings distinguish $n[0] \oplus \{0, 1, 2, 3, 4, 5, 6, 7\}$ from $n[0] \oplus i$ for all other i. At this point I guessed that the timings for any key k would leak $k[5] \oplus \{0, 1, 2, 3, 4, 5, 6, 7\}$, i.e., would leak the top five bits of $k[5]$. This guess later turned out to be correct.

Similar analyses showed useful patterns for every value of b. Correlations identified $n[0] \oplus \{0, 1, \ldots, 7\}$; $n[1]$; $n[2] \oplus \{0, 1, \ldots, 7\}$; $n[3] \oplus \{0, 1, \ldots, 7\}$; $n[4] \oplus \{0, 1, 2, 3\}$; $n[5] \oplus \{0, 1, \ldots, 7\}$; $n[6] \oplus \{0, 1, \ldots, 7\}$; $n[7] \oplus \{0, 1, \ldots, 7\}$; $n[8] \oplus \{0, 1, \ldots, 7\}$; $n[9]$; $n[10] \oplus \{0, 1, \ldots, 7\}$; $n[11] \oplus \{0, 1, \ldots, 7\}$; $n[12] \oplus \{0, 1, \ldots, 7\}$;

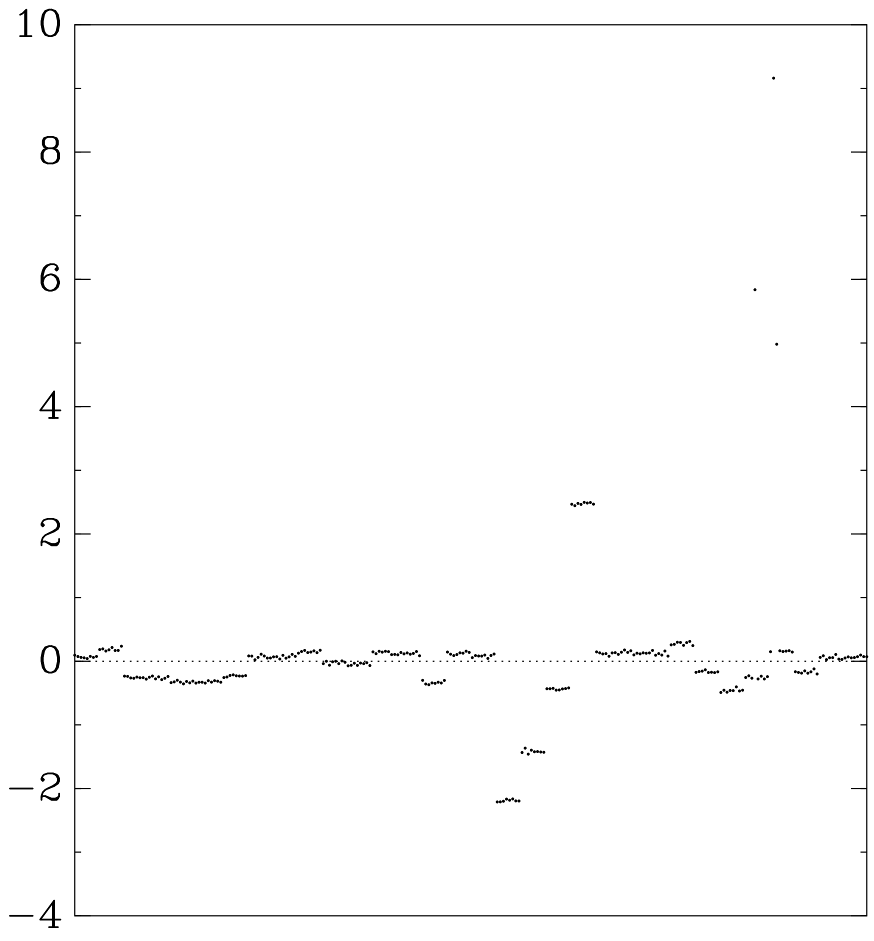

Fig. 5.3. How $n[15]$ affects OpenSSL AES timings for $k = 0$ on a Pentium III inside the targeted server. The horizontal axis is the choice of $n[15]$, from 0 through 255. The vertical axis is the average number of cycles for about 2^{22} 800-byte packets with that choice of $n[15]$, minus the average number of cycles for all choices of $n[15]$.

$n[13]$; $n[14] \oplus \{0, 1, \ldots, 7\}$; and $n[15] \oplus \{0, 1, \ldots, 7\}$. The correlations for $b = 6$ and $b = 12$ were only about 1 squared cycle but were still visible.

After inspecting these results for 400-byte packets, I killed the `study` program and then tried again with 800-byte packets:

```
% ./study 192.168.123.141 800 > study.800
```

After the same number of packets, correlations identified $n[0] \oplus \{0, 1, \ldots, 7\}$; $n[1] \oplus \{0, 1, \ldots, 7\}$; $n[2] \oplus \{0, 1, \ldots, 7\}$; $n[3]$; $n[4] \oplus \{0, 1, 2, 3\}$; $n[5] \oplus \{0, 1, \ldots, 7\}$; $n[6] \oplus \{0, 1, \ldots, 7\}$; $n[7] \oplus \{0, 1, \ldots, 7\}$; $n[8] \oplus \{0, 1, \ldots, 7\}$; $n[9] \oplus \{0, 1\}$; $n[10] \oplus \{0, 1, \ldots, 7\}$; $n[11] \oplus \{0, 1, \ldots, 7\}$; $n[12] \oplus \{0, 1, \ldots, 7\}$; $n[13] \oplus \{0, 1, \ldots, 7\}$; $n[14] \oplus \{0, 1\}$; and $n[15]$. See, for example, Figure 5.3, showing a clear maximum for $n[15] = 225$. The timing deviation for 800-byte packets with any particular $n[b]$ was only about 45 cycles, compared to 130 cycles for 400-byte packets.

I also tried 600-byte packets. The results were similar, this time pinpointing $n[4]$, $n[8]$, and $n[12]$. The timing deviation was again about 45 cycles.

6 Carrying out the attack

In the victim's role, on an 850MHz Pentium III desktop computer running FreeBSD 4.8, I took 16 bytes from `/dev/urandom`, FreeBSD's cryptographic pseudorandom number generator:

```
% dd if=/dev/urandom of=secretkey bs=16 count=1
1+0 records in
1+0 records out
16 bytes transferred in 0.000113 secs (141580 bytes/sec)
```

I then compiled and ran the server:

```
% gcc --version
2.95.4
% openssl version
OpenSSL 0.9.7a Feb 19 2003
% gcc -O3 -o server server.c -lcrypto
% printenv | wc -c
     549
% ./server 192.168.123.58 < secretkey
```

At this point the server was listening for UDP packets sent to port 10000 of `192.168.123.58`. It was using a secret AES key: the bytes from `/dev/urandom`.

I then switched to the attacker's role and moved to the attacker's computer. I sent one packet to the victim's server, using the `ciphertext` program shown in Appendix C, to see the server's scrambled zero:

```
% gcc -O3 -o ciphertext ciphertext.c
% ./ciphertext 192.168.123.58 > attack
% cat attack
32 6b 9e f2 94 17 1b 46 a3 55 8b 59 6c 06 fa 7f
```

I then sent random 800-byte packets to the victim's server:

```
% ./study 192.168.123.58 800 > attack.800
```

After 2^{25} random packets—about 160 minutes, at a rate of a few thousand packets per second—I correlated the resulting timings with the known timings collected in Section 5, using the `correlate` program shown in Appendix D:

```
% gcc -O3 -o correlate correlate.c -lm
% (tail -4096 study.800; tail -4096 attack.800) \
| ./correlate >> attack
```

The `correlate` program prints, in hexadecimal, the $n[b]$ offsets producing large correlations:

```
 24  0    e3 e2 e5 e4 e6 d3 e0 d6 d2 e7 e1 d1 d5 d4 d7 d0 ...
 16  1    72 77 76 75 71 70 74 73 99 98 9a 9e 9b 9d 9c 9f
  8  2    81 87 83 86 84 85 82 80
  1  3    a9
  4  4    8b 8a 89 88
  8  5    b4 b1 b0 b2 b5 b7 b6 b3
 31  6    66 67 62 65 64 60 63 61 ed e9 ef ec e8 eb ee ea ...
  8  7    a7 a0 a1 a5 a2 a6 a4 a3
  8  8    b9 bb b8 ba be bf bd bc
  8  9    7f 7e 7c 7d 79 78 7a 7b
  8 10    d3 d6 d1 d5 d4 d2 d7 d0
  8 11    3a 3f 3d 3b 39 3c 3e 38
176 12    ef ec eb e9 ea ed ee e8 3f 39 3b 0b 3c 0c 0f 38 ...
 16 13    ec e8 ee ea eb e9 ef ed 94 96 91 95 90 97 92 93
  8 14    3c 3d 3f 38 39 3e 3b 3a
  1 15    35
```

I guessed at this point that the secret $k[3]$ was 169 (hexadecimal `a9`); that the secret $k[7]$ was either 167, 160, 161, 165, 162, 166, 164, or 163; etc. I then tried 600-byte packets:

```
% ./study 192.168.123.58 600 > attack.600
```

I computed correlations after 2^{25} random packets:

```
% (tail -4096 study.600; tail -4096 attack.600) \
| ./correlate >> attack
```

I then did the same for 2^{27} random 400-byte packets. These extra packet sizes narrowed the range of possibilities for various key bytes, and in particular pinned down $k[1]$, $k[4]$, $k[8]$, $k[9]$, $k[12]$, and $k[13]$:

```
1  4    89
1  8    bb
1 12    9e
1  1    74
1  9    7f
2 13    ec e3
```

Presumably trying more packets, and more packet sizes, would have imposed further constraints on k, but at this point I decided that I had collected more than enough timing information. I searched through the possible keys using the extremely crude `search` program shown in Appendix E, checking each possible key against the server's scrambled zero:

```
% gcc -O3 -o search search.c -lcrypto
% ./search < attack
The key is 43 74 84 a9 89 b1 62 a7 bb 7f d0 3d 9e ec 3d 35.
```

This took one minute, about 2^{37} Athlon cycles. Replacing zero with harder-to-predict plaintext would not have stopped the attack: any recognizable plaintext would have allowed a similar search to pinpoint the key.

At this point I declared success. I knew an AES key that scrambled zero in the same way as the server's AES key, and that had the same apparent timing characteristics; obviously this string was the server's key.

To recap the effect of the attack: The server's AES key had not been used in any way other than to scramble data using the OpenSSL AES implementation. I had not peeked at the key; the only way I had accessed the key was by talking to the server through the network. I had nevertheless deduced the server's key.

Finally, to confirm success, I switched back to the victim's role, moved back to the victim's desktop computer, and peeked at the key. It matched.

Future work

This area offers many obvious opportunities for cryptanalytic research:

- Start guessing keys as soon as the set of *most likely* keys is small enough, rather than waiting for the set of *possible* keys to be small enough. I never performed a careful computation of key-byte probabilities; I wouldn't be surprised if the first 2^{20} 800-byte and 600-byte packets were sufficient to identify the key.
- Consider interactions between b's, so that the time variability for one b does not add noise to other b's; for example, model the time for n as $\sum_b t(b, n[b])$. This would reduce the number of packets required.
- Instead of choosing $n[b]$ uniformly for each packet, gradually adapt the $n[b]$ distribution to focus on the choices farthest from average. This would further reduce the number of packets required.
- By considering table lookups in (e.g.) the last AES round, develop an attack that works with highly structured AES inputs (as in, e.g., counter mode) rather than random-looking AES inputs (as in, e.g., CBC).
- Apply a cycle-accurate CPU simulator to the compiled server code to see how $n[b]$ is influencing the AES timings; explain why a particular $n[b] \oplus k[b]$ value maximizes the timings. (Sections 8 through 15 of this paper have some relevant information, but they are aimed at the limited problem of creating high-speed constant-time AES code; they are not meant to explain all of the timing variations in non-constant-time AES code.)
- By carefully reviewing memory accesses in the server and in the surrounding operating system, figure out packets that will force targeted AES S-box lines to be kicked out of L2 cache. This will drastically increase the input-dependent variability in AES timings, and will correspondingly reduce the number of packets required to identify $k[b]$.
- Apply the same attack to other CPUs and other servers. The difficulty of writing constant-time high-speed AES software—see Sections 8 through 15 of this paper—means that it would be surprising if any AES-based servers are immune.

I do not intend to pursue any of these research directions. As far as I'm concerned, it is already blazingly obvious that the AES time variability is a huge security threat. We need constant-time cryptographic software! Once we've all switched to constant-time software, we won't care about the exact difficulty of attacking variable-time software.

7 Errors in the AES standardization process

"RAM cache hits can produce timing characteristics," Kocher wrote in 1996 in [16, Section 11], after explaining in detail how to extract secret keys from timing leaks in another operation. A decade later, we still have cryptographic software—in particular, AES libraries—whose timing obviously varies with, among other things, input-dependent RAM cache hits. How did this happen?

Was the National Institute of Standards and Technology unaware of timing attacks during the development of AES? No. In its "Report on the development of the Advanced Encryption Standard," NIST spent several pages discussing side-channel attacks, specifically timing attacks and power attacks. It explicitly considered the difficulty of defending various operations against these attacks. For example, NIST stated in [19, Section 5.1.5] that MARS was "difficult to defend" against these attacks.

Did NIST decide, after evaluating timing attacks, that those attacks were unimportant? No. Exactly the opposite occurred, as discussed below.

So what went wrong? Answer: NIST failed to recognize that table lookups do not take constant time. "Table lookup: not vulnerable to timing attacks," NIST stated in [19, Section 3.6.2]. NIST's statement was, and is, incorrect.

NIST went on to consider the slowness of AES implementations designed to protect against side-channel attacks. For example, NIST stated that providing "some defense" for MARS meant "severe performance degradation." NIST stated in [19, Section 5.3.5] that Rijndael gained a "major speed advantage over its competitors when such protections are considered." This statement was based directly on the incorrect notion that table lookups take constant time. NIST made the same comment in its "summary assessments of the finalists," and again in its concluding paragraph explaining the selection of Rijndael as AES. See [19, Section 6.5] and [19, Section 7].

NIST was not the first to make these comments. In a paper "Resistance against implementation attacks: a comparative study of the AES proposals," the Rijndael designers commented that "a comparative study of the efficiency of the different algorithms on platforms that allow the described attacks should take into account the measures to be taken to thwart these attacks"; incorrectly stated that table lookup "is not susceptible to a timing attack"; and concluded that Rijndael was "favorable," i.e., "relatively easy to secure." See [9, Section 5], [9, Section 3.3], and [9, Section 4.12].

In response to this paper, one of the Rijndael designers characterized cache-timing attacks as "irrelevant for cryptographic design." This characterization is out of whack with what actually occurred in the design of the Advanced

Encryption Standard. Cryptographic designers should—and the AES designers did, at great length—consider the difficulty of protecting against timing attacks. Where the AES designers erred was in asserting that table lookups take constant time.

It is hard to guess what might have happened if this error had been pointed out during the AES standardization process. *None* of the fifteen AES candidates provide high performance on all popular general-purpose CPUs—except by using instructions with input-dependent timings, notably S-box lookups.

8 The quest for constant-time high-speed AES software

It is reasonably easy to write AES software that takes constant time on today's popular general-purpose CPUs; here **constant** means "independent of the AES key and input." The quantities $S[b]$ and $S'[b]$ in Section 2 can be expressed as fairly short formulas using constant-time bit operations: xor, constant-distance shift, etc.

The resulting software is immune to timing attacks. Unfortunately, it is also *much* slower than AES software using S-boxes.

The remaining sections of this paper focus on the problem of building AES software that takes constant time—and is therefore immune to timing attacks—*without* sacrificing so much speed. Why do AES timings depend on inputs? What is the least expensive way for the AES implementor to eliminate each of these dependencies?

It turns out that there are a surprising number of leaks. See Sections 9 through 15. It also turns out that some of the leaks are very difficult to plug. See, in particular, Sections 13 and 15.

Is constant time overkill?

Some cryptographers presume that input-dependent cryptographic timings are not fatal. Perhaps the input dependence is incomprehensible to the attacker.

The problem is that input-dependent timings often *are* fatal. Allowing input-dependent AES timings means throwing away the confidence that has been gained from repeated failures to cryptanalyze the AES outputs. Perhaps the cryptanalyst can see a pattern in the AES timings without seeing any pattern in the AES outputs.

In contrast, proactively writing constant-time AES software guarantees that timing attacks on the AES software are as difficult as pure cryptanalysis of the AES function.

An intermediate approach is to try to randomize the information being leaked. This is a common response to hardware side channels; see, e.g., [6]. Unfortunately, most randomization techniques are guaranteed safe only against the simplest attacks. Furthermore, the AES randomization techniques appear to produce software even slower than a simple sequence of constant-time bit operations.

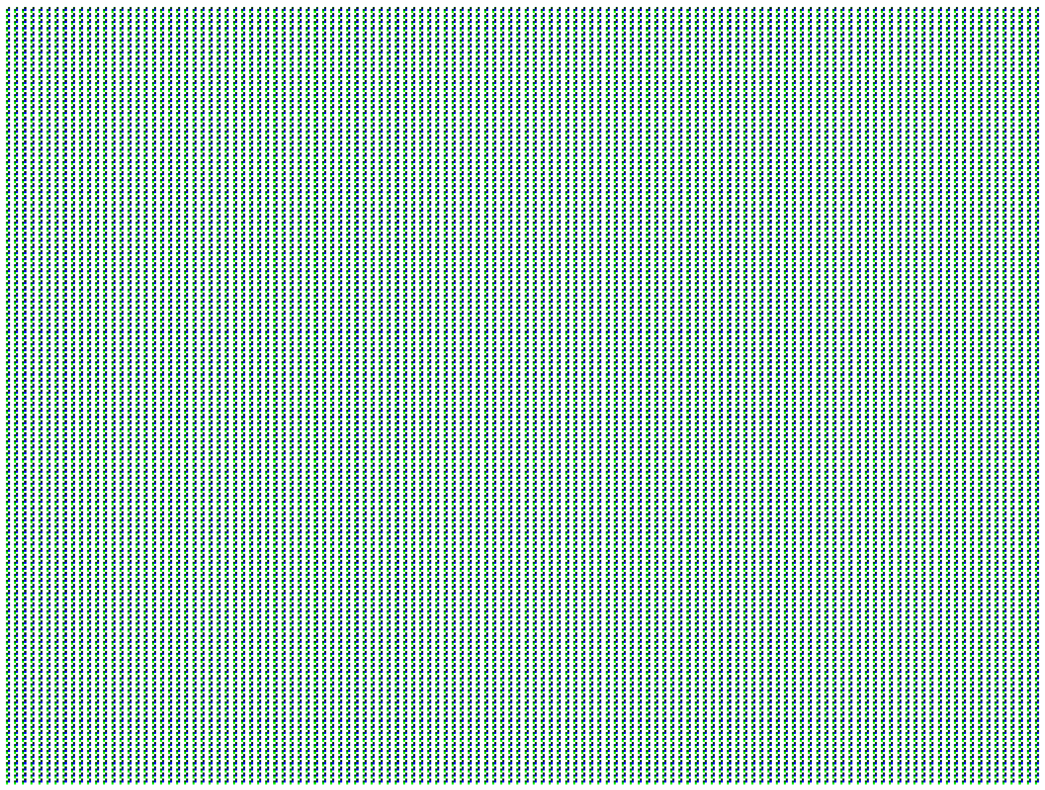

Fig. 8.1. AES timings, using the author's AES software, for 128 keys and 128 inputs on a Pentium M. This picture is a 128×128 array of blocks. There is one key for each row of blocks; there is one input for each column of blocks. The pattern of colors in a block reflects the distribution of timings for that (key, input) pair. Each block in this picture is visually identical. Reader beware: this picture is easily corrupted by low-resolution displays.

How can constant time be recognized?

Suppose that we are given several implementations of AES. How do we tell which implementations actually take constant time?

One answer is to collect timings and look for input-dependent patterns. For example, one can repeatedly measure the time taken by AES for one (key, input) pair, convert the distribution of timings into a small block of colors, and then repeat for many keys and inputs. Constant-time AES software would have the same block of colors for every key and input, as in Figure 8.1. A glance at Figure 8.2 shows instantly that the AES implementation in OpenSSL does not take constant time. See `http://cr.yp.to/mac/variability1.html` and `http://cr.yp.to/mac/variability2.html` for many more of these pictures.

Unfortunately, even if software passes this test, there is no guarantee that it takes constant time. For example, the problem described in Section 14 depends on the layout of variables in memory, and can easily be missed by a test that uses only one layout—no matter how many inputs are considered by the test.

Fig. 8.2. OpenSSL AES timings for 128 keys and 128 inputs on a Pentium M. This picture is a 128 × 128 array of blocks. There is one key for each row of blocks; there is one input for each column of blocks. The pattern of colors in a block reflects the distribution of timings for that (key, input) pair. There is tremendous variability among blocks in this picture. Reader beware: this picture is easily corrupted by low-resolution displays.

A tester who hears about this problem can build a test that checks for it; but there may be other problems that the tester has not heard about.

Let me emphasize that I am *not* asserting that the software in Figure 8.1 takes constant time. On the contrary: I am saying that this sort of testing *cannot* provide a reasonable level of confidence that software takes constant time.

Another answer is to perform a detailed review of the amount of time taken by each target CPU for each instruction in the computation. Unfortunately, most CPU manufacturers do a terrible job of documenting their chips. Consider, for example, the Opteron cache-bank conflicts documented in [2, Section 5.8] and discussed in Section 15 of this paper. Experiment shows that the original Athlon has the same conflicts—but, as far as I know, this fact appears nowhere in any AMD documentation. Furthermore, my workaround for this time variability relies on several undocumented Athlon instruction-decoding details.

Advice for CPU designers

CPU manufacturers should thoroughly document the performance of their chips. In particular, they need to highlight every variation in their instruction timings, and to guarantee that there are no other variations. As this paper demonstrates, hidden CPU performance information is a security threat.

(Hidden CPU performance information is also a quite severe annoyance for programmers writing high-speed code. Some CPU manufacturers, evidently aware of the large effect that these programmers have on CPU market share, have taken the sensible step of publishing accurate CPU simulators. Other CPU manufacturers seem to be under the delusion that they are capable of keeping secrets from their multi-billion-dollar competitors.)

CPU designers should also consider adding AES support to their instruction sets. For example, a CPU could support fast constant-time instructions that compute $T_0[b]$, $T_1[b]$, $T_2[b]$, and $T_3[b]$ given b. The requisite circuit area is very small and easily justified by the current benefits for AES users.

9 Problem: Skipping an operation is faster than doing it

Input-dependent branches—performing a computation for some inputs and not for others—are the most obvious source of input-dependent timings, and are featured throughout the literature on timing attacks.

More than three decades ago, the TENEX operating system compared a user-supplied string against a secret password one character at a time, stopping at the first difference. Attackers determined the stopping point—and thus, after a few hundred tries, the secret password—by monitoring the time taken by the comparison. See, e.g., [8].

Kocher in [16] pointed out that some current cryptographic software was vulnerable to timing attacks through—among other things—the extra time taken by large-integer subtractions performed for some inputs and not others.

(After hearing about Kocher's paper, I began consciously writing constant-time cryptographic software. Example 1: Several years ago I issued a public warning that the microSPARC-IIep has data-dependent FPU timings. Example 2: My NIST P-224 elliptic-curve point-multiplication software uses powering to compute reciprocals modulo a prime, in part because Euclid's algorithm has input-dependent timings; see [5, page 20]. Example 3: When I explained cache-timing attacks in a course on cryptography in 1997, I categorically recommended against all use of S-boxes in new cryptographic designs. I realize that this view is not yet shared by the entire cryptographic community.)

Koeune and Quisquater in [17] presented details of a fast timing attack on a "careless implementation" of AES that used input-dependent branches on an embedded CPU. "The result presented here is not an attack against Rijndael, but against bad implementations of it," they wrote in [17, Section 7].

In Section 2 of this paper, the values $S'[b]$ are $2S[b]$ if $S[b] < 128$, or $2S[b] \oplus 283$ if $S[b] \geq 128$. Koeune and Quisquater were considering AES implementations that, given b, computed $S[b]$ (somehow), then $2S[b]$, and finally $2S[b] \oplus 283$, but branched around the $\oplus 283$ if $S[b] < 128$. One branch-free alternative is to compute `X=S>>7; X|=(X<<1); X|=(X<<3); Sprime=(S<<1)^X` in C notation.

This problem does not arise in high-speed AES software for common general-purpose CPUs. On-the-fly computation of S and S' would be unacceptably slow. The values $T_0[b]$, $T_1[b]$, $T_2[b]$, $T_3[b]$ are precomputed; the AES computation is a straight-line sequence of byte extractions, T_i table lookups, and $\oplus$ operations. Unfortunately, as discussed in subsequent sections, the table lookups do not take constant time.

10 Problem: Cache is faster than DRAM

Recently used **lines** of memory are automatically saved in a limited-size **cache**. The Athlon, for example, divides memory into 64-byte lines; there are 327680 bytes of cache containing 5120 recently used lines. Similarly, a typical Pentium III divides memory into 32-byte lines, and has 524288 bytes of cache containing 16384 recently used lines.

The second most obvious source of input-dependent timings is that reading from a cached line takes less time than reading from an uncached line. For example, if $T_0[0]$ is cached while $T_0[64]$ is not, then reading $T_0[b]$ will take less time for $b = 0$ than for $b = 64$, leaking information about b.

This type of leak was briefly mentioned in [16, Section 11], [14, Section 7], and [10, Section 16.3]. Page in [20], and then Tsunoo, Saito, Suzaki, Shigeri, and Miyauchi in [22], presented fast timing attacks on DES when all S-boxes were out of cache before each DES call. Tsunoo et al. mentioned a strategy for attacking out-of-cache AES if many (unusual) “plaintexts with long encryption time” could be found; see [22, Section 4.1].

Advice for AES implementors

The obvious way to avoid this leak is to guarantee that all of the AES S-boxes are in cache throughout each AES computation. But this is much more easily said than done; and, even if it is done, there are still timing leaks! There are four specific problems:

- AES S-box lines may be kicked out of cache to make room for memory lines used by computations other than AES. Consequently, accessing all the AES S-box lines at the beginning of a program is not sufficient. Periodically accessing all the AES S-box lines is also not sufficient; lines can be kicked out of cache very quickly, as discussed in Section 12.
- Accessing all the AES S-boxes *before every AES computation*, as suggested in [20, Section 5.1] and [22, Section 5], is much more expensive and is *still* not sufficient. AES S-box lines may be kicked out of cache by the AES computation itself. See Section 12.

- Even if the AES computation fits entirely within cache, the AES computation may be interrupted—and S-box lines may be kicked out of cache—by other operations. See Section 13.
- Keeping all AES S-box lines in cache does *not* guarantee constant-time S-box lookups. See Sections 11, 14, and 15.

Detailed advice regarding these problems appears in subsequent sections.

There is another problem when the AES S-boxes are not forced into cache until immediately before the AES computation. Consider, for example, simple code to load one entry from each S-box line. If the lines were not already in cache then the loads may take hundreds of cycles. During this time, the CPU (even a nominally "in-order" CPU such as the UltraSPARC) may begin the AES computation, and conceivably may even begin the first AES S-box lookup, even though the S-boxes are not yet all in cache. One can try to force all pending loads to complete by, for example, storing and reloading some data, but CPU documentation sometimes fails to guarantee that such techniques actually work. The safest approach is to ensure that there are enough cycles between the initial S-box loads and the AES computation.

Advice for CPU designers

An extra CPU instruction would solve all of these problems, not just for AES but for any computation that needs constant-time S-box lookups. See Section 11.

11 Problem: L1 cache is faster than L2 cache

Modern CPUs actually have two (and occasionally more) levels of cache. All recently used lines of **level-2 cache** are automatically saved in (or moved to) a limited-size **level-1 cache**. Reading from a line in L1 cache takes less time than reading from a line in L2 cache.

On the Athlon, for example, there are 65536 bytes of L1 cache containing 1024 recently used lines, and 262144 bytes of L2 cache containing 4096 other recently used lines. The result of a load from L1 cache is available after 3 cycles; the result of a load from L2 cache is available after, typically, 11 cycles.

Similarly, a typical Pentium III has 16384 bytes of L1 cache containing 512 recently used lines, and 524288 bytes of L2 cache containing 16384 recently used lines, including all the lines in L1 cache.

Some other chips: The Pentium 4, like the Pentium 1, has 8192 bytes of L1 cache. The PowerPC G4 has 32768 bytes of L1 cache. The PowerPC RS64 IV has 131072 bytes of L1 cache. The UltraSPARC II has 16384 bytes of L1 cache. The UltraSPARC III has 65536 bytes of L1 cache.

Advice for AES implementors

The obvious way to avoid this leak is to guarantee that all of the AES S-boxes are in L1 cache throughout each AES computation. But, as in Section 10, there are four problems:

- AES S-box lines may be kicked out of L1 cache to make room for memory lines used by computations other than AES.
- AES S-box lines may be kicked out of L1 cache by the AES computation itself.
- AES S-box lines may be kicked out of L1 cache by other operations that interrupt the AES computation.
- Keeping all AES S-box lines in L1 cache does *not* guarantee constant-time S-box lookups.

See Sections 12, 13, 14, and 15 for further discussion.

Advice for CPU designers

CPUs could solve these time-variability problems by adding an L1-table-lookup instruction. The instruction would (1) ensure that an entire table is in L1 cache and (2) load a selected table entry in a constant number of CPU cycles. Of course, no timing guarantees are provided for table sizes that do not fit into L1 cache.

A 32-bit instruction can easily name an output register, a register pointing to the start of the table, an index register, a scale (among a few possible values), and a table size (among several possible values). But it seems more natural to use just one bit in the instruction, with special registers pointing to the start and end of the table. Typical applications would keep the registers constant for quite a few instructions.

The most straightforward implementation of the instruction is as follows. Check that each line of the table is in L1 cache; if not, wait for the missing lines to appear in L1 cache. Then load the requested data. Then set a bit saying that the check can be skipped next time—making the next L1-table-lookup instruction as fast as a normal load. Clear the bit whenever the special registers change or any line is evicted from L1 cache.

This instruction is subtly different from a lock-into-cache instruction. It does not prevent eviction—except while it is executing. It can safely be exposed to unprivileged code.

12 Problem: Cache associativity is limited

The Athlon's L1 cache is **2-way associative**. This means that each line of memory has only 2 locations in L1 cache where it can be placed; all lines with the same address modulo 32768 are competing for those 2 locations. If three lines with the same address modulo 32768 are read, the first line is booted out of the L1 cache.

Typical AES software uses several different arrays: the AES input; the key (perhaps expanded); the AES output; the stack, for temporary variables; and the AES S-boxes. The positions of these arrays can easily overlap modulo 32768. Consequently, even if the AES software starts by making sure that the AES S-boxes are in L1 cache, it might lose some S-box lines from cache by accessing (e.g.) the key and the stack.

Similarly, a very small amount of code between two AES computations will kick an AES S-box line out of L1 cache if that code happens to access a few lines at matching positions modulo 32768.

The Pentium III's L1 cache is 4-way associative: all lines with the same address modulo 4096 are competing for 4 locations in the L1 cache. The 4-way associativity means that four small arrays can never kick each other out of cache; on the other hand, random positions are much more likely to collide modulo 4096 than modulo 32768.

The Pentium 4's L1 cache is 4-way associative. The PowerPC G4's L1 cache is 8-way associative. The PowerPC RS64 IV's L1 cache is 2-way associative. The UltraSPARC II's L1 cache is 1-way associative ("direct mapped"). The UltraSPARC III's L1 cache is 4-way associative.

Advice for AES implementors

The simplest way to prevent variables from kicking each other out of cache is to control the positions of those variables in memory. For example, if a network packet, an AES input, an AES key, an AES output, and the AES S-boxes are adjacent in memory, and if they are collectively small enough to fit into L1 cache, then they will not kick each other out of cache.

A useful space-saving technique is to compress the tables T_0, T_1, T_2, T_3 into a single 2048-byte table with 8-byte entries of the form

$$(S'[b], S[b], S[b], S[b] \oplus S'[b], S'[b], S[b], S[b], 0).$$

Observe that $T_0[b]$, $T_1[b]$, $T_2[b]$, $T_3[b]$ start at, respectively, the first, fourth, third, and second bytes of this entry. Some CPUs do not allow a single instruction to quickly load 4-byte quantities from addresses not divisible by 4, but the Athlon, Pentium, and PowerPC do allow it, and on most of these CPUs there is no speed penalty when the 8-byte entries do not cross 8-byte memory boundaries.

If an AES input and key must be loaded from uncontrolled locations then those loads must be presumed to have kicked some AES S-box lines out of cache. Similarly, any uncontrolled code—even a very small amount of code, such as a simple library call—must be presumed to have kicked some AES S-box lines out of cache. The AES S-boxes must be loaded into L1 cache before the next AES computation.

13 Problem: Code can be interrupted

Assume that the AES S-boxes are in L1 cache at the beginning of the AES computation, and that nothing in the AES computation kicks any S-box lines out of L1 cache. Does this guarantee that the S-boxes are in L1 cache throughout the entire computation? No! The problem is that the AES computation might pause in the middle to allow another program to run.

This type of problem was briefly mentioned by Tsunoo et al. in [22, Section 5] ("clear a cache during the encryption"). Osvik and Tromer have announced full key recovery using this type of attack against a server on the same computer; the announcement came in February 2005, a few months after the initial publication of this paper, but the attack was developed independently. The Osvik-Tromer attack has the amusing feature of recovering some key bits without receiving any data directly from the AES computation: it detects AES cache misses through timings of another process rather than timings of the AES computation.

Here are three examples of how the AES computation might be interrupted:

- The network supplies an incoming packet, or the disk supplies a block of data, demanding immediate attention from the operating system. The operating system's network driver or disk driver is likely to kick S-box lines out of L1 cache.
- A CPU timer interrupts this process, giving the operating system a chance to run another process that has been waiting for a while to use the CPU. Even if the operating system does not switch to another process, the operating system itself can kick S-box lines out of L1 cache.
- Exploited by Osvik and Tromer: A Pentium 4 CPU with "hyperthreading" decides, during a cycle when the AES computation is not using all of the CPU's resources, to follow a pending instruction from another process. That instruction might kick an S-box line out of L1 cache.

Advice for AES implementors

Pentium 4 hyperthreading can easily be disabled by the computer owner. AES implementors should encourage computer owners to disable hyperthreading.

Other process interruptions are much more difficult to handle. A process on a single-CPU system can detect any serious interruption by checking the CPU's cycle counter before and after the AES computation; unfortunately, by the time the interruption has been detected, the damage has already been done!

One solution is to incorporate cryptographic software into the operating-system kernel. The kernel can disable interrupts during the AES computation. Most CPUs have emergency interrupts that cannot be disabled, but the kernel can, and typically does, handle those by shutting down the computer.

I am not aware of any other solutions. As far as I can tell, current systems do not provide any way for non-kernel code to guarantee constant-time S-box lookups.

Additional kernel code is usually a security threat: every line of kernel code has complete control over the computer and needs to be carefully reviewed for security problems. But this is not a reason to keep cryptographic functions such as AES out of the kernel. AES software needs to be carefully reviewed for security problems whether or not it is in the kernel.

Advice for CPU designers

CPUs and operating systems obviously cannot allow unprivileged code to disable interrupts, but they could allow unprivileged code to take different action upon return from an interrupt. AES software could then set an interrupt return address that restarts the AES computation. If the computation is restarted repeatedly (because interrupts are occurring at surprisingly high frequency), the software could switch to a slow computation without S-boxes.

A more efficient alternative is for CPUs to put a specified table back into L1 cache upon interrupt return. Of course, the same effect is achieved by the L1-table-lookup instruction proposed in Section 11; the instruction is restarted upon interrupt return, if it is interruptible in the first place.

14 Problem: Stores can interfere with loads

Assume that all AES S-boxes are in L1 cache throughout the AES computation. Does this guarantee constant-time S-box lookups? No!

On some CPUs, a load from L1 cache takes slightly more time if it involves the same set of cache lines as a recent store to L1 cache. For example, on the Pentium III, a load from L1 cache takes slightly more time if it involves the same cache line modulo 4096 as a recent store to L1 cache. I learned this effect from [12, Section 14.7], and mentioned it briefly in the first version of this paper. Intel in [3, page 2.43] has documented the same effect for the Pentium M.

As far as I know, all previous AES software for the Pentium III stores data to the stack during the main loop. The relevant stack locations can overlap S-box lines modulo 4096; they are guaranteed to do so if the AES tables are 4096 consecutive bytes, for example. The stores then interfere with loads from those S-box lines, causing a detectable input-dependent timing burp.

Let me emphasize this point: the time taken to load an array element can depend on the array index *even if all array entries are in L1 cache*. Keeping the AES S-boxes in L1 cache throughout the AES computation does *not* guarantee that AES S-box lookups take constant time.

Advice for AES implementors

AES software for the Pentium can avoid load-store conflicts as follows. Use compressed 2048-byte S-boxes as described in Section 12. At the beginning of the AES computation, shift the stack to a position just below the beginning of the AES S-boxes modulo 4096. Here are the relevant instructions at the top and bottom of my `aes_ppro` software:

```
aes_ppro:
eax = esp
eax -= aes_ppro_constants
eax &= 4096
eax += 208
```

```
    esp -= eax
    ...
    esp += eax
    return
```

After the initial adjustment of esp, the 208 bytes esp[0],esp[1],...,esp[207] are just before the 2048-byte AES tables (aes_ppro_constants) modulo 4096, so stores to those stack positions do not interfere with table loads.

15 Problem: Cache-bank throughput is limited

Assume that all AES S-boxes are in L1 cache throughout the AES computation, and that the AES software avoids the load-store interference described in Section 14. Does this guarantee constant-time S-box lookups? No! On some CPUs, loads from L1 cache can bump into each other, taking slightly different amounts of time depending on their addresses.

On the Athlon, each 64-byte cache line is divided into 8 **banks**. Usually the L1 cache can perform two loads from L1 cache every cycle. However, if the two loads are from bank 0 of two lines with different addresses modulo 32768, or from bank 1 of two such lines, etc., then the second load will wait for a cycle.

This paper began when I wrote some Athlon AES software and noticed this effect in the AES timings. Thanks to Andreas Kaiser for drawing my attention to [2, Section 5.8], which documents the same effect for the Opteron. I published the first version of this paper on 2004.11.11.

This timing variability is not specific to the Athlon line of CPUs. For example, the Pentium 1 has similar cache-bank conflicts. See [12, Section 10.2].

Advice for AES implementors

There are two approaches to eliminating this timing leak on the Athlon. The first approach is to control which banks are used. One can, as an extreme, use 64-byte spacing between T_i entries, so that a load from T_i always uses the same bank, independent of the array index. But this slows down the computation, for two reasons:

- Multiplying an array index by 64 takes an extra instruction compared to multiplying it by 1, 2, 4, or 8.
- The tables are spread across eight times as many cache lines as before, so forcing the tables into cache takes many more instructions.

The large tables also make it more difficult to place variables as described in Sections 12 and 14.

The second approach is to arrange for each load to take place in a separate cycle. The Athlon spreads instructions across three execution units, each of which can perform at most one load per cycle; my current aes_athlon software appears to successfully eliminate all cache-bank conflicts by putting all of the loads into a single execution unit. I handled the instruction spreading as follows:

- Arrange instructions in groups of three, with each load instruction at the beginning of a group. The idea is that the three instructions will be assigned to the three execution units in order. There are, however, many exceptions that have to be avoided.
- Do not allow instruction groups to cross a 16-byte boundary in memory. The AMD manuals do not make clear how instructions are assigned to execution units when this rule is violated. (I have heard a simple model for this, based on a vague comment from AMD, but I know from experiment that the model is incorrect.) Following this rule often requires inserting no-op groups of various lengths, as discussed below.
- Use `0x90` for a 1-byte no-op, `0x66 0x90` for a 2-byte no-op, `lea (%esp),%esp` for a 3-byte no-op, and `lea 0(%esp),%esp` for a 4-byte no-op. Some other no-ops are too complicated for the Athlon to handle in a group.
- To save CPU time, insert instruction prefixes when possible to eliminate no-op groups. Most instructions are unaffected by an `0x3e` prefix (data segment) or an `0x2e 0x3e` prefix (code segment, data segment); experiment indicates that the Athlon can handle two 1-byte prefixes on each instruction in a group.

A similar approach will work for the Pentium 1. The Pentium 1 has completely different (and much better documented) instruction-scheduling rules, but the basic idea is the same: never try two loads in a single cycle.

Other CPUs must be carefully reviewed for the types of timing variance described in this section and the previous section, and for any other input-dependent timings. Every new CPU poses a potential new challenge. Even if my code successfully runs in constant time on an Athlon, for example, it might not run in constant time on an Opteron or Athlon 64.

References

1. —, *Secure hash standard*, Federal Information Processing Standard 180-2, National Institute of Standards and Technology, Washington, 2002. URL: `http://csrc.nist.gov/publications/fips/`.
2. —, *Software optimization guide for AMD Athlon 64 and AMD Opteron processors*, Advanced Micro Devices, 2004. URL: `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF`.
3. —, *IA-32 Intel architecture optimization: reference manual*, Intel Corporation, 2004. URL: `http://www.intel.com/design/pentium4/manuals/index_new.htm`.
4. Paulo S. L. M. Barreto, *The AES block cipher in C++* (2003). URL: `http://planeta.terra.com.br/informatica/paulobarreto/EAX++.zip`.
5. Daniel J. Bernstein, *A software implementation of NIST P-224* (2001). URL: `http://cr.yp.to/talks.html#2001.10.29`.
6. Johannes Bloemer, Jorge Guajardo Merchan, Volker Krummel, *Provably secure masking of AES* (2004). URL: `http://eprint.iacr.org/2004/101/`.
7. David Brumley, Dan Boneh, *Remote timing attacks are practical* (2003). URL: `http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf`.
8. Larry Campbell, *Tenex hackery* (1991). URL: `http://groups-beta.google.com/group/alt.folklore.computers/msg/00d243bb0caa9f69?dmode=source`.

9. Joan Daemen, Vincent Rijmen, *Resistance against implementation attacks: a comparative study of the AES proposals* (1999). URL: `http://csrc.nist.gov/CryptoToolkit/aes/round1/pubcmnts.htm`.
10. Niels Ferguson, Bruce Schneier, *Practical cryptography*, Wiley, 2003. ISBN 0471223573.
11. Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, Tadayoshi Kohno, *Helix: fast encryption and authentication in a single cryptographic primitive*, in [13] (2003), 330–346. URL: `http://www.macfergus.com/helix/`.
12. Agner Fog, *How to optimize for the Pentium family of microprocessors*, 2004. URL: `http://www.agner.org/assem/`.
13. Thomas Johansson (editor), *Fast software encryption: 10th international workshop, FSE 2003, Lund, Sweden, February 24–26, 2003, revised papers*, Lecture Notes in Computer Science, 2887, Springer-Verlag, Berlin, 2003. ISBN 3–540–20449–0.
14. John Kelsey, Bruce Schneier, David Wagner, Chris Hall, *Side channel cryptanalysis of product ciphers*, Journal of Computer Security **8** (2000), 141–158. ISSN 0926–227X.
15. Neal Koblitz (editor), *Advances in cryptology—CRYPTO '96*, Lecture Notes in Computer Science, 1109, Springer-Verlag, Berlin, 1996.
16. Paul C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, in [15] (1996), 104–113. URL: `http://www.cryptography.com/timingattack/paper.html`.
17. François Koeune, Jean-Jacques Quisquater, *A timing attack against Rijndael* (1999). URL: `http://www.dice.ucl.ac.be/crypto/techreports.html`.
18. David A. McGrew, John Viega, *The security and performance of the Galois/counter mode of operation* (2004). URL: `http://eprint.iacr.org/2004/193/`.
19. James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, Edward Roback, *Report on the development of the Advanced Encryption Standard (AES)*, Journal of Research of the National Institute of Standards and Technology **106** (2001). URL: `http://nvl.nist.gov/pub/nistpubs/jres/106/3/cnt106-3.htm`.
20. Daniel Page, *Theoretical use of cache memory as a cryptanalytic side-channel* (2002). URL: `http://eprint.iacr.org/2002/169/`.
21. Bart Preneel (editor), *Fast software encryption: second international workshop, Leuven, Belgium, 14–16 December 1994, proceedings*, Lecture Notes in Computer Science, 1008, Springer-Verlag, Berlin, 1995. ISBN 3–540–60590–8.
22. Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, Hiroshi Miyauchi, *Cryptanalysis of DES implemented on computers with cache*, in [23] (2003), 62–76.
23. Colin D. Walter, Cetin K. Koc, Christof Paar (editors), *Cryptographic hardware and embedded systems—CHES 2003*, Springer-Verlag, Berlin, 2003. ISBN 3–540–40833–9.
24. David J. Wheeler, Roger M. Needham, *TEA, a tiny encryption algorithm*, in [21] (1995), 363–366.

A Appendix: `server.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <openssl/aes.h>


unsigned int timestamp(void)
{
  unsigned int bottom;
  unsigned int top;
  asm volatile(".byte 15;.byte 49" : "=a"(bottom),"=d"(top));
  return bottom;
}


unsigned char key[16];
AES_KEY expanded;
unsigned char zero[16];
unsigned char scrambledzero[16];

void handle(char out[40],char in[],int len)
{
  unsigned char workarea[len * 3];
  int i;

  for (i = 0;i < 40;++i) out[i] = 0;
  *(unsigned int *) (out + 32) = timestamp();

  if (len < 16) return;
  for (i = 0;i < 16;++i) out[i] = in[i];

  for (i = 16;i < len;++i) workarea[i] = in[i];
  AES_encrypt(in,workarea,&expanded);
  /* a real server would now check AES-based authenticator, */
  /* process legitimate packets, and generate useful output */

  for (i = 0;i < 16;++i) out[16 + i] = scrambledzero[i];
  *(unsigned int *) (out + 36) = timestamp();
}
```

```
struct sockaddr_in server;
struct sockaddr_in client; socklen_t clientlen;
int s;
char in[1537];
int r;
char out[40];

main(int argc,char **argv)
{
  if (read(0,key,sizeof key) < sizeof key) return 111;
  AES_set_encrypt_key(key,128,&expanded);
  AES_encrypt(zero,scrambledzero,&expanded);

  if (!argv[1]) return 100;
  if (!inet_aton(argv[1],&server.sin_addr)) return 100;
  server.sin_family = AF_INET;
  server.sin_port = htons(10000);

  s = socket(AF_INET,SOCK_DGRAM,0);
  if (s == -1) return 111;
  if (bind(s,(struct sockaddr *) &server,sizeof server) == -1)
    return 111;

  for (;;) {
    clientlen = sizeof client;
    r = recvfrom(s,in,sizeof in,0
                 ,(struct sockaddr *) &client,&clientlen);
    if (r < 16) continue;
    if (r >= sizeof in) continue;
    handle(out,in,r);
    sendto(s,out,40,0,(struct sockaddr *) &client,clientlen);
  }
}
```

B Appendix: `study.c`

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <poll.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

double packets;
double ttotal;
double t[16][256];
double tsq[16][256];
long long tnum[16][256];
double u[16][256];
double udev[16][256];

char n[16];

void tally(double timing)
{
  int j;
  int b;
  for (j = 0;j < 16;++j) {
    b = 255 & (int) n[j];
    ++packets;
    ttotal += timing;
    t[j][b] += timing;
    tsq[j][b] += timing * timing;
    tnum[j][b] += 1;
  }
}
```

```
int s;
int size;

void studyinput(void)
{
  int j;
  char packet[2048];
  char response[40];
  struct pollfd p;

  for (;;) {
    if (size < 16) continue;
    if (size > sizeof packet) continue;
    /* a mediocre PRNG is sufficient here */
    for (j = 0;j < size;++j) packet[j] = random();
    for (j = 0;j < 16;++j) n[j] = packet[j];
    send(s,packet,size,0);
    p.fd = s;
    p.events = POLLIN;
    if (poll(&p,1,100) <= 0) continue;
    while (p.revents & POLLIN) {
      if (recv(s,response,sizeof response,0) == sizeof response) {
        if (!memcmp(packet,response,16)) {
          unsigned int timing;
          timing = *(unsigned int *) (response + 36);
          timing -= *(unsigned int *) (response + 32);
          if (timing < 10000) { /* clip tail to reduce noise */
            tally(timing);
            return;
          }
        }
      }
      if (poll(&p,1,0) <= 0) break;
    }
  }
}
```

```
void printpatterns(void)
{
  int j;
  int b;
  double taverage;

  taverage = ttotal / packets;

  for (j = 0;j < 16;++j)
    for (b = 0;b < 256;++b) {
      u[j][b] = t[j][b] / tnum[j][b];
      udev[j][b] = tsq[j][b] / tnum[j][b];
      udev[j][b] -= u[j][b] * u[j][b];
      udev[j][b] = sqrt(udev[j][b]);
    }

  for (j = 0;j < 16;++j) {
    for (b = 0;b < 256;++b) {
      printf("%2d %4d %3d %lld %.3f %.3f %.3f %.3f\n"
        ,j
        ,size
        ,b
        ,tnum[j][b]
        ,u[j][b]
        ,udev[j][b]
        ,u[j][b] - taverage
        ,udev[j][b] / sqrt(tnum[j][b])
        );
    }
  }
  fflush(stdout);
}

int timetoprint(long long inputs)
{
  if (inputs < 10000) return 0;
  if (!(inputs & (inputs - 1))) return 1;
  return 0;
}
```

```
int main(int argc,char **argv)
{
  struct sockaddr_in server;
  long long inputs = 0;

  if (!argv[1]) return 100;
  if (!inet_aton(argv[1],&server.sin_addr)) return 100;
  server.sin_family = AF_INET;
  server.sin_port = htons(10000);

  if (!argv[2]) return 100;
  size = atoi(argv[2]);

  while ((s = socket(AF_INET,SOCK_DGRAM,0)) == -1) sleep(1);
  while (connect(s,(struct sockaddr *) &server,sizeof server
              ) == -1) sleep(1);

  for (;;) {
    studyinput();
    ++inputs;
    if (timetoprint(inputs)) printpatterns();
  }
}
```

C Appendix: `ciphertext.c`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <poll.h>

int main(int argc,char **argv)
{
  struct sockaddr_in server;
  int s;
  char packet[40];
  char response[40];
  struct pollfd p;
  int j;
  if (!argv[1]) return 100;
  if (!inet_aton(argv[1],&server.sin_addr)) return 100;
  server.sin_family = AF_INET;
  server.sin_port = htons(10000);
  while ((s = socket(AF_INET,SOCK_DGRAM,0)) == -1) sleep(1);
  while (connect(s,(struct sockaddr *) &server,sizeof server
              ) == -1) sleep(1);
  for (;;) {
    for (j = 0;j < sizeof packet;++j) packet[j] = random();
    send(s,packet,sizeof packet,0);
    p.fd = s;
    p.events = POLLIN;
    if (poll(&p,1,100) <= 0) continue;
    while (p.revents & POLLIN) {
      if (recv(s,response,sizeof response,0) == sizeof response) {
        if (!memcmp(packet,response,16)) {
          for (j = 0;j < 16;++j)
            printf("%02x ",255 & (unsigned int) response[j + 16]);
          printf("\n");
          return 0;
        }
      }
      if (poll(&p,1,0) <= 0) break;
    }
  }
}
```

D Appendix: `correlate.c`

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double t[16][256];
double tdev[16][256];
double u[16][256];
double udev[16][256];

void readdata(void)
{
  int lines;
  int b;
  int size;
  int j;
  long long packets;
  double cycles;
  double deviation;
  double aboveaverage;
  double avdev;

  for (lines = 0;lines < 8192;++lines) {
    if (scanf("%d%d%d%lld%lf%lf%lf%lf"
             ,&b
             ,&size
             ,&j
             ,&packets
             ,&cycles
             ,&deviation
             ,&aboveaverage
             ,&avdev
            ) != 8) exit(100);
    b &= 15;
    j &= 255;
    if (lines < 4096) {
      t[b][j] = aboveaverage;
      tdev[b][j] = avdev;
    } else {
      u[b][j] = aboveaverage;
      udev[b][j] = avdev;
    }
  }
}
```

```
double c[256];
double v[256];
int cpos[256];
int cposcmp(const void *v1,const void *v2)
{
  int *i1 = (int *) v1;
  int *i2 = (int *) v2;
  if (c[255 & *i1] < c[255 & *i2]) return 1;
  if (c[255 & *i1] > c[255 & *i2]) return -1;
  return 0;
}

void processdata(void)
{
  int b;
  int i;
  int j;
  int numok;
  double z;
  for (b = 0;b < 16;++b) {
    for (i = 0;i < 256;++i) {
      c[i] = v[i] = 0;
      cpos[i] = i;
      for (j = 0;j < 256;++j) {
        c[i] += t[b][j] * u[b][i ^ j];
        z = tdev[b][j] * u[b][i ^ j];
        v[i] += z * z;
        z = t[b][j] * udev[b][i ^ j];
        v[i] += z * z;
      }
    }
    qsort(cpos,256,sizeof(int),cposcmp);
    numok = 0;
    for (i = 0;i < 256;++i)
      if (c[cpos[0]] - c[cpos[i]] < 10 * sqrt(v[cpos[i]]))
        ++numok;
    printf("%3d %2d   ",numok,b);
    for (i = 0;i < 256;++i)
      if (c[cpos[0]] - c[cpos[i]] < 10 * sqrt(v[cpos[i]]))
        printf(" %02x",cpos[i]);
    printf("\n");
  }
}
```

```
int main()
{
  readdata();
  processdata();
  return 0;
}
```

E Appendix: search.c

```c
#include <openssl/aes.h>
#include <string.h>
#include <stdio.h>

unsigned char zero[16];
unsigned char scrambledzero[16];
unsigned char ciphertext[16];
double weight[16][256];
double maxweight[16];
unsigned char key[16];
AES_KEY expanded;

void doit(int b)
{
  int x;
  if (b == 16) {
    AES_set_encrypt_key(key,128,&expanded);
    AES_encrypt(zero,scrambledzero,&expanded);
    if (!memcmp(scrambledzero,ciphertext,16)) {
      printf("The key is");
      for (x = 0;x < 16;++x) printf(" %02x",key[x]);
      printf(".\n");
    }
    return;
  }
  for (x = 0;x < 256;++x)
    if (weight[b][x] == maxweight[b]) {
      key[b & 15] = x;
      doit(b + 1);
    }
}
```

```
int main()
{
  int b;
  int x;
  int n;
  int i;

  for (b = 0;b < 16;++b) {
    scanf("%x",&x);
    ciphertext[b] = x;
  }
  while (scanf("%d",&n) == 1) {
    scanf("%d",&b);
    for (i = 0;i < n;++i) {
      scanf("%x",&x);
      weight[b & 15][x & 255] += 1.0 / n;
    }
  }
  for (b = 0;b < 16;++b)
    for (x = 0;x < 256;++x)
      if (weight[b][x] > maxweight[b])
        maxweight[b] = weight[b][x];

  doit(0);

  return 0;
}
```