Reconstructing RSA Private Keys from Random Key Bits

Nadia Heninger[1] and Hovav Shacham[2]

[1] Princeton University
nadiah@cs.princeton.edu
[2] University of California, San Diego
hovav@cs.ucsd.edu

Abstract. We show that an RSA private key with small public exponent can be efficiently recovered given a 0.27 fraction of its bits at random. An important application of this work is to the "cold boot" attacks of Halderman et al. We make new observations about the structure of RSA keys that allow our algorithm to make use of the redundant information in the typical storage format of an RSA private key. Our algorithm itself is elementary and does not make use of the lattice techniques used in other RSA key reconstruction problems. We give an analysis of the running time behavior of our algorithm that matches the threshold phenomenon observed in our experiments.

1 Introduction

In this paper, we present a new algorithm for the problem of reconstructing RSA private keys given a random δ-fraction of their bits. For RSA keys with small public exponent, our algorithm reconstructs the private key with high probability when $\delta \geq 0.27$. The runtime analysis of our algorithm relies on an assumption (Conjecture 1) and is thus heuristic; but we have verified experimentally that it succeeds with high probability.

Motivation: cold boot attacks. An important application of our algorithm is key recovery from the randomly distributed unidirectional bit corruption observed in the recent work of Halderman et al. [10], which demonstrated that DRAM remanence effects make possible practical, nondestructive attacks that recover (a degraded version of) secret keys stored in a computer's memory. Using these "cold boot" attacks, attackers with physical access to a machine can break popular disk encryption systems or recover an SSL server's private key.

One consequence of the nature of the attack is that a perfect image of the contents of memory may not be available to the attacker; instead, some bits may have been flipped. Halderman et al. observe that, within a DRAM region, the decay is overwhelmingly either 0 → 1 or 1 → 0. The decay direction for a region can be determined by comparing the number of 0s and 1s. (In an uncorrupted key we expect these to be approximately equal.) For a region of 1 → 0 decay, a 1 bit in the decayed version is known (with high probability) to correspond to a

1 bit in the original key, whereas a 0 bit might correspond to either a 0 or 1 bit in the original key. If a ρ fraction of bits decays and 0s and 1s were present in equal numbers in the key then we will know, given the degraded representation, a $\delta = (1 - \rho)/2$ fraction of key bits.

Halderman et al. further showed that it is possible to exploit redundancy in key data to create algorithms for reconstructing DES, AES, and cipher tweak keys from their degraded in-memory representations. In addition, they experimented with reconstructing RSA keys by using the public modulus N to correct its partly-known factors p and q. We extend this idea to take into account other fields of an RSA private key and provide an analysis of the resulting algorithm's runtime behavior. Our improvement makes a significant difference in practice: their algorithm takes several minutes to recover a 2048-bit RSA key from 12% unidirectional corruption; ours takes under a second to recover a 2048-bit key from as much as 46% unidirectional corruption.

Our algorithm and its performance. Our two main results in this paper are: (1) an algorithm for reconstructing RSA private keys given a random δ-fraction of their bits; and (2) an analysis of the algorithm's runtime behavior for random inputs that shows that it will succeed in expected quadratic time when $\delta \geq .27$. The runtime analysis depends crucially on both a uniformly random distribution of known bits and the assumption that the effect of a bit error during reconstruction is propagated uniformly through subsequent bits of the key.

Our algorithm performs better than the algorithm given by Halderman et al. because it is able to make use of five components of the RSA private key: p, q, d, d_p, and d_q. We can use known bits in d, d_p, and d_q to make progress where bits in p and q are not known. To relate d to the rest of the private key, we make use of techniques due to Boneh, Durfee, and Frankel [4]; to relate d_p and d_q to the rest of the private key, we make new observations about the structure of RSA keys that may be of independent interest. This is discussed in Section 2.

If the algorithm has access to fewer components of the RSA private key, the algorithm will still perform well given a sufficiently large fraction of the bits. For example, it can efficiently recover a key given

$\delta = .27$ fraction of the bits of p, q, d, d_p, and d_q.
$\delta = .42$ fraction of the bits of p, q, and d.
$\delta = .57$ fraction of the bits of p and q.

The reconstruction algorithm itself, described in Section 3, is elementary and does not make use of the lattice basis reduction or integer programming techniques that have been applied to other kinds of RSA key reconstruction problems. At each step, it branches to explore all possible keys, and prunes these possibilities using our understanding of the structure of RSA keys and the partial information we are given about key bits. We give an analysis of the algorithm for random inputs in Section 4. We obtain a sharp threshold around $2 - 2^{(4/5)} \approx 27\%$ of known key bits. Below this threshold, the expected number of keys examined is exponential in the number of bits of the key, and above this threshold, the expected number of keys examined is close to linear. Note that this threshold applies only to our particular approach. We suspect these results could be improved using more sophisticated methods.

Finally, we have implemented our algorithm and performed extensive experiments using it. The results are described in Section 5. The algorithm's observed behavior matches our analytically derived bounds and validates the heuristic assumptions made in the analysis.

Small public-exponent RSA. Our algorithm is specialized to the case where *the public exponent e is small.* The small-e case is, for historical reasons, the overwhelmingly common one in deployed RSA applications such as SSL/TLS. For example, until recently Internet Explorer would reject TLS server certificates with an RSA public exponent longer than 32 bits [5, p. 8]. The choice $e = 65537 = 2^{16} + 1$ is especially widespread. Of the certificates observed in the UCSD TLS Corpus [23] (which was obtained by surveying frequently-used TLS servers), 99.5% had $e = 65537$, and all had e at most 32 bits.

Related work. Inspired by cold boot attacks, Akavia, Goldwasser, and Vaikuntanathan [1] formally introduced *memory attacks*, a class of side-channel attacks in which the adversary is leaked a (shrinking) function of the secret key. One research direction, pursued by Akavia, Goldwasser, and Vaikuntanathan and, in followup work, Naor and Segev [18], is constructing cryptosystems provably secure against memory attacks.[3] Another research direction is to evaluate the security of existing cryptosystems against memory attacks. Our work is along this latter direction.

There is a great deal of work on both factoring and reconstructing RSA private keys given a fraction of the bits.

Maurer [14] shows that integers can be factored in polynomial time given oracle access to an ϵ fraction of the bits of a factor.

In a slightly stricter model, the algorithm has access to a fixed subset of consecutive bits of the integer factors or RSA private keys. Rivest and Shamir [21] first solved the problem for a 2/3-fraction of the least significant bits of a factor using integer programming. This was improved to 1/2 of the least or most significant bits of a factor using lattice-reduction techniques pioneered by Coppersmith [6]; we refer the reader surveys by Boneh [3] and May [16] as well as May's Ph. D. thesis [15] for bibliographies. More recently, Herrmann and May extended these techniques to efficiently factor given at most $\log \log N$ known blocks of bits [12].

The problem we seek to solve can be viewed as a further relaxation of the conditions on access to the key bits to a fully random subset. These lattice-reduction techniques are not directly applicable to our problem because they rely on recovering *consecutive* bits of the key (expressed as small integer solutions to modular equations), whereas the missing bits we seek to find are randomly distributed

[3] There has been substantial other recent work on designing cryptosystems secure in related key-leakage models (e.g., [20, 8, 2]); for a survey, see Goldwasser's invited talk at Eurocrypt 2009 [9] and the references therein.

throughout the degraded keys. It is possible to express our reconstruction problem as a knapsack, and there are lattice techniques for solving knapsack problems (see, e.g., Nguyen and Stern [19]), but we have not managed to improve on our solution by this approach.

2 RSA Private Keys

The PKCS#1 standard specifies [22, Sect. A.1.2] that an RSA private key include at least the following information:

- the (n-bit) modulus N and public exponent e;
- the private exponent d;
- the prime factors p and q of N;
- d modulo $p-1$ and $q-1$, respectively denoted d_p and d_q; and
- the inverse of q modulo p, denoted q_p^{-1}.

In practice, an RSA key in exactly this format can be recovered from the RAM of a machine running Apache with OpenSSL [10]. The first items – N and e – make up the public key and are already known to the attacker. A naïve RSA implementation would use d to perform the private-key operation $c \mapsto c^d \bmod N$, but there is a more efficient approach, used by real-world implementations such as OpenSSL, that is enabled by the remaining private-key entries. In this approach, one computes the answer modulo p and q as $(c \bmod p)^{d_p}$ and $(c \bmod q)^{d_q}$, respectively; then combines these two partial answers by means of q_p^{-1} and the Chinese Remainder Theorem (CRT). This approach requires two exponentiations but of smaller numbers, and is approximately four times as fast as the naïve method [17, p. 613].

Observe that the information included in PKCS#1 private keys is *highly redundant.* In fact, knowledge of any single one of p, q, d, d_p, and d_q is sufficient to reveal the factorization of N.[4] It is this redundancy that we will use in reconstructing a corrupted RSA key.

We now derive relations between p, q, d, d_p, and d_q that will be useful in mounting the attack. The first such relation is obvious:

$$N = pq \ . \tag{1}$$

Next, since d is the inverse of e modulo $\varphi(N) = (p-1)(q-1) = N - p - q + 1$, we have

$$ed \equiv 1 \pmod{\varphi(N)}$$

and, modulo $p-1$ and $q-1$,

$$ed_p \equiv 1 \pmod{p-1} \qquad \text{and} \qquad ed_q \equiv 1 \pmod{q-1} \ .$$

[4] This is obvious for p and q and well known for d (cf. [7]); d_p reveals p as $\gcd(a^{ed_p-1} - 1, N)$ with high probability for random a provided $d_p \neq d_q$, and similarly for d_q; if d_p and d_q are equal to each other then they are also equal to d.

As it happens, it is more convenient for us to write explicitly the terms hidden in the three congruences above, obtaining

$$ed = k(N - p - q + 1) + 1 \tag{2}$$
$$ed_p = k_p(p - 1) + 1 \tag{3}$$
$$ed_q = k_q(q - 1) + 1 \ . \tag{4}$$

It may appear that we have thereby introduced three new unknowns: k, k_p, and k_q. But in fact for small e we can compute each of these three variables given even a badly-degraded version of d.

Computing k. The following argument, due to Boneh, Durfee, and Frankel [4], shows that k must be in the range $0 < k < e$. We know $d < \varphi(N)$. Assume $e \leq k$; then $ed < k\varphi(N) + 1$, which contradicts (2). The case $k = 0$ is also impossible, as can be seen by reducing (2) modulo e. This shows that we can enumerate all possible values of k, having assumed that e is small.

For each such choice k', define

$$\tilde{d}(k') \overset{\text{def}}{=} \left\lfloor \frac{k'(N+1)+1}{e} \right\rfloor \ .$$

As Boneh, Durfee, and Frankel observe, when k' equals k, this gives an excellent approximation for d:

$$0 \leq \tilde{d}(k) - d \leq k(p+q)/e < p + q \ .$$

In particular, when p and q are balanced, we have $p + q < 3\sqrt{N}$, which means that $\tilde{d}(k)$ agrees with d on their $\lfloor n/2 \rfloor - 2$ most significant bits. (Our analysis applies also in the less common case when p and q are unbalanced, but we omit the details.) This means that small-public-exponent RSA leaks half the bits of the private exponent in one of the candidate values $\tilde{d}(1), \ldots, \tilde{d}(e-1)$.

The same fact allows us to go in the other direction, using information about d to determine k, as again noted by Boneh, Durfee, and Frankel. We are given $\tilde{d}$, a corrupted version of d. We enumerate $\tilde{d}(1), \ldots, \tilde{d}(e-1)$ and check which of these agrees, in its more significant half, with the known bits of $\tilde{d}$. Provided that $\delta n/2 \gg \lg e$, there will be just one value of k' for which $\tilde{d}(k')$ matches; that value is k. Even for 1024-bit N and 32-bit e, there is, with overwhelming probability, enough information to compute k for any δ we consider in this paper. This observation has two implications:

1. we learn the correct k used in (2); and
2. we correct the more significant half of the bits of $\tilde{d}$, by copying from $\tilde{d}(k)$.

Computing k_p and k_q. Once we have determined k, we can compute k_p and k_q. First, observe that by an analysis like that above, we can show that $0 < k_p, k_q < e$. This, of course, means that $k_p = (k_p \bmod e)$ and $k_q = (k_q \bmod e)$; when we solve for k_p and k_q modulo e, this will reveal the actual values used in

(3) and (4). Now, reducing equations (1)–(4) modulo e, we obtain the following congruences:

$$\begin{aligned} N &\equiv pq && (5)\\ 0 &\equiv k(N-p-q+1)+1 && (6)\\ 0 &\equiv k_p(p-1)+1 && (7)\\ 0 &\equiv k_q(q-1)+1 \ . && (8) \end{aligned}$$

These are four congruences in four unknowns: p, q, k_p, and k_q; we solve them as follows. From (7) and (8) we write $(p-1) \equiv -1/k_p$ and $(q-1) \equiv -1/k_q$; we substitute these into the equation obtained from using (5) to reexpress $\varphi(N)$ in (6): $0 \equiv k(N-p-q+1)+1 \equiv k(p-1)(q-1)+1 \equiv k(-1/k_p)(-1/k_q)+1 \equiv k/(k_pk_q)+1$, or

$$k + k_pk_q \equiv 0 \ . \qquad (9)$$

Next, we return to (6), substituting in (7), (8), and (9):

$$\begin{aligned} 0 &\equiv k(N-p-q+1)+1 \\ &\equiv k(N-1) - k(p-1+q-1)+1 \\ &\equiv k(N-1) - (-k_pk_q)(-1/k_p - 1/k_q)+1 \\ &\equiv k(N-1) - (k_q+k_p)+1 \ ; \end{aligned}$$

we solve for k_p by substituting $k_q = -k/k_p$, obtaining

$$0 \equiv k(N-1) - (k_p - k/k_p) + 1 \ ,$$

or, multiplying both sides by k_p and rearranging,

$$k_p^2 - \bigl[k(N-1)+1\bigr]k_p - k \equiv 0 \ . \qquad (10)$$

This congruence is easy to solve modulo e and, in the common case where e is prime, has two solutions, just as it would over $\mathbb{C}$. One of the two solutions is the correct value of k_p; and it is easy to see, by symmetry, that the other must be the correct value of k_q. We need therefore try just two possible assignments to k_p and k_q in reconstructing the RSA key. When e has m distinct prime factors, there may be up to 2^m roots [4].

Note that we also learn the values of p and q modulo e. If we then use the procedure outlined below to decode the r least significant bits of p (up to a list of possibilities), we will know $p \bmod e2^r$; we can then factor N, provided $r + \lg e > n/4$, by applying Boneh, Durfee, and Frankel's Corollary 2.2 ([4]; a generalization of Coppersmith's attack on RSA with known low-order bits [6, Theorem 5] that removes the restriction that the partial knowledge of p must be modulo a power of 2).

3 The Reconstruction Algorithm

Once we have the above relationships between key data, the remainder of the attack consists of enumerating all possible partial keys and pruning those that do not satisfy these constraints. More precisely, given bits 1 through $i-1$ of a potential key, generate all combinations of values for bit i of p, q, d, d_p, d_q, and keep a candidate combination if it satisfies (1), (2), (3), and (4) mod 2^i.

The remainder of this section details how to generate and prune these partial solutions.

In what follows, we assume that we know the values of k_p and k_q. When equation (10) has two distinct solutions, we must run the algorithm twice, once for each of the possible assignments to k_p and k_q.

Let $p\,[i]$ denote the ith bit of p, where the least significant bit is bit 0, and similarly index the bits of q, d, d_p and d_q. Let $\tau(x)$ denote the exponent of the largest power of 2 that divides x.

As p and q are large primes, we know they are odd, so we can correct $p\,[0] = q\,[0] = 1$. It follows that $2 \mid p-1$, so $2^{1+\tau(k_p)} \mid k_p(p-1)$. Thus, reducing (3) modulo $2^{1+\tau(k_p)}$, we have

$$ed_p \equiv 1 \pmod{2^{1+\tau(k_p)}} \ .$$

Since we know e, this allows us immediately to correct the $1+\tau(k_p)$ least significant bits of d_p. Similar arguments using (4) and (2) allow us to correct the $1+\tau(k_q)$ and $2+\tau(k)$ bits of d_q and d, respectively.

What is more, we can easily see that, having fixed bits $< i$ of p, a change in $p\,[i]$ affects d_p not in bit i but in bit $i+\tau(k_p)$; and, similarly, a change in $q\,[i]$ affects $d_q\,\bigl[i+\tau(k_q)\bigr]$, and a change in $p\,[i]$ or $q\,[i]$ affects $d\,\bigl[i+\tau(k)\bigr]$. When any of k, k_p, or k_q is odd, this is just the trivial statement that changing bit i of the right-hand side of an equation changes bit i of the left-hand side. Powers of 2 in k_p shift left the bit affected by $p\,[i]$, and similarly for the other variables.

Having recovered the least-significant bits of each of our five variables, we now attempt to recover the remaining bits. For each bit index i, we consider a slice of bits:

$$p\,[i] \qquad q\,[i] \qquad d\,\bigl[i+\tau(k)\bigr] \qquad d_p\,\bigl[i+\tau(k_p)\bigr] \qquad d_q\,\bigl[i+\tau(k_q)\bigr] \ .$$

For each possible solution up to bit slice $i-1$, generate all possible solutions up to bit slice i that agree with that solution at all but the ith position. If we do this for all possible solutions up to bit slice $i-1$, we will have enumerated all possible solutions up to bit slice i. Above, we already described how to obtain the only possible solution up to $i=0$; this is the solution we use to start the algorithm. The factorization of N will be revealed in one or more of the possible solutions once we have reached $i = \lfloor n/2 \rfloor$.[5]

All that remains is how to lift a possible solution (p', q', d', d'_p, d'_q) for slice $i-1$ to possible solutions for slice i. Naïvely there are $2^5 = 32$ such possibilities, but in fact there are at most 2 and, for large enough δ, almost always fewer.

First, observe that we have four constraints on the five variables: equations (1), (2), (3), and (4). By plugging in the values up to slice $i-1$, we obtain

[5] In fact, as we discussed in Section 2 above, information sufficient to factor N will be revealed much earlier, at $i = \lceil n/4 - \lg e \rceil$.

from each of these a constraint on slice i, namely values $c_1, \ldots, c_4$ such that the following congruences hold modulo 2:

$$\begin{aligned} p\left[i\right] + q\left[i\right] &\equiv c_1 \pmod 2 \\ d\left[i + \tau(k)\right] + p\left[i\right] + q\left[i\right] &\equiv c_2 \pmod 2 \\ d_p\left[i + \tau(k_p)\right] + p\left[i\right] &\equiv c_3 \pmod 2 \\ d_q\left[i + \tau(k_q)\right] + q\left[i\right] &\equiv c_4 \pmod 2 \ . \end{aligned} \tag{11}$$

For example, if N and $p'q'$ agree at bit i, $c_1 = 0$; if not, $c_1 = 1$. Four constraints on five unknowns means that there are exactly two possible choices for bit slice i satisfying these four constraints. (Expressions for the c_is are given in (13).)

Next, it may happen that we know the correct value of one or more of the bits in the slice, through our partial knowledge of the private key. These known bits might agree with neither, one, or both of the possibilities derived from the constraints above. If neither possible extension of a solution up to $i - 1$ agrees with the known bits, that solution is pruned. If δ is sufficiently large, the number of possibilities at each i will be kept small.

4 Algorithm Runtime Analysis

The main result of this section is summarized in the following informal theorem.

Theorem 1. *Given the values of a $\delta = .27$ fraction of the bits of p, q, d, $d \bmod p$, and $d \bmod q$, the algorithm will correctly recover an n-bit RSA key in expected $O(n^2)$ time with probability $1 - \frac{1}{n^2}$.*

The running time of the algorithm is determined by the number of partial keys examined. To bound the total number of keys seen by the program, we will first understand how the structure of the constraints on the RSA key data determines the number of partial solutions generated at each step of the algorithm. Then we will use this understanding to calculate some of the distribution of the number of solutions generated at each step over the randomness of p and q and the missing bits. Finally we characterize the global behavior of the program and provide a bound on the probability that the total number of branches examined over the entire run of the program is too large.

Lifting solutions mod 2^i. The process of generating bit i of a partial solution given bits 0 through $i-1$ can be seen as lifting a solution to the constraint equations mod 2^i to a solution mod 2^{i+1}. Hensel's lemma characterizes the conditions when this is possible.

Lemma 1 (Multivariate Hensel's Lemma). *A root $\mathbf{r} = (r_1, r_2, \ldots, r_n)$ of the polynomial $f(x_1, x_2, \ldots, x_n) \bmod \pi^i$ can be lifted to a root $\mathbf{r} + \mathbf{b} \bmod \pi^{i+1}$ if $\mathbf{b} = (b_1\pi^i, b_2\pi^i, \ldots, b_n\pi^i)$, $0 \le b_j \le \pi - 1$ is a solution to the equation*

$$f(\mathbf{r} + \mathbf{b}) = f(\mathbf{r}) + \sum_j b_j \pi^i f_{x_j}(\mathbf{r}) \equiv 0 \pmod{\pi^{i+1}} \ .$$

(Here, f_{x_j} is the partial derivative of f with respect to x_j.)

We can rewrite the lemma using the notation of Section 3. Write $\mathbf{r}$ in base $\pi = 2$ and assume the i first bits are known. Then the lemma tells us that the next bit of $\mathbf{r}$, $\mathbf{r}[i] = (r_1[i], r_2[i], \ldots)$, must satisfy

$$f(\mathbf{r})[i] + \sum_j f_{x_j}(\mathbf{r}) r_j[i] \equiv 0 \pmod 2 \ . \tag{12}$$

In our case, the constraint polynomials generated in Section 2, equations (1)–(4) form four simultaneous equations in five variables. Given a partial solution (p', q', d', d'_p, d'_q) up to slice i of the bits, we apply the condition in equation (12) above to each polynomial and reduce modulo 2 to obtain the following conditions, modulo 2, on bit i:

$$\begin{aligned} p\,[i] + q\,[i] &\equiv (n - p'q')\,[i] \\ d\,\big[i + \tau(k)\big] + p\,[i] + q\,[i] &\equiv \big(k(N+1) + 1 - k(p' + q') - ed'\big)\,\big[i + \tau(k)\big] \\ d_p\,\big[i + \tau(k_p)\big] + p\,[i] &\equiv \big(k_p(p' - 1) + 1 - ed'_p\big)\,\big[i + \tau(k_p)\big] \\ d_q\,\big[i + \tau(k_q)\big] + q\,[i] &\equiv \big(k_q(q' - 1) + 1 - ed'_q\big)\,\big[i + \tau(k_q)\big] \ . \end{aligned} \tag{13}$$

These are precisely (11).

4.1 Local branching behavior

Without additional knowledge of the keys, the system of equations in (13) is underconstrained, and each partial satisfying assignment can be lifted to two partial satisfying assignments for slice i. If bit $i-1$ of a variable x is known, the corresponding $x\,[i-1]$ is fixed to the value of this bit, and the new partial satisfying assignments correspond to solutions of (13) with these bit values fixed. There can be zero, one, or two new solutions at bit i generated from a single solution at bit $i-1$, depending on the known values.

Now that we have a framework for characterizing the partial solutions generated at step i from a partial solution generated at step $i-1$, we will assume that a random fraction δ of the bits of the key values are known, and estimate the expectation and variance of the number of these solutions that will be generated.

In order to understand the number of solutions to the equation, we would like to understand the behavior of the c_i when the partial solution may not be equal to the real solution. Let $\Delta x = x - x'$, then substituting $x' = x - \Delta x$ into (13) we see that any solution to (11) corresponds to a solution to

$$\begin{aligned} \Delta p\,[i] + \Delta q\,[i] &\equiv (q\Delta p + p\Delta q + \Delta p \Delta q)\,[i] && \pmod 2 \\ \Delta d\,\big[i + \tau(k)\big] + \Delta p\,[i] + \Delta q\,[i] &\equiv (e\Delta d + k\Delta p + k\Delta q))\,\big[i + \tau(k)\big] && \pmod 2 \\ \Delta d_p\,\big[i + \tau(k_p)\big] + \Delta p\,[i] &\equiv (e\Delta d_p - k_p\Delta p)\,\big[i + \tau(k_p)\big] && \pmod 2 \\ \Delta d_q\,\big[i + \tau(k_q)\big] + \Delta q\,[i] &\equiv (e\Delta d_q - k_q\Delta q)\,\big[i + \tau(k_q)\big] && \pmod 2 \end{aligned}$$

and $\Delta x\,[i]$ is restricted to 0 if bit i of x is fixed.

Incorrect solutions generated from a correct solution. When the partial satisfying assignment is correct, all of the $\varDelta x$ will be equal to 0. If all of the $\varDelta x\,[i]$ are unconstrained or if only $\varDelta d\,[i+\tau(k)]$ is set to 0, there will be two possible solutions (of which we know one is "good" and the other is "bad"), otherwise there will be a single good solution. Let Z_g be a random variable denoting the number of bad solutions at bit $i+1$ generated from a single good solution at bit i. Since each $\varDelta x\,[i]$ is set to 0 independently with probability δ, the expected number of bad solutions generated from a good solution is equal to

$$\mathrm{E}\, Z_g = \delta(1-\delta)^4 + (1-\delta)^5 \qquad \text{and} \qquad \mathrm{E}\, Z_g^2 = \mathrm{E}\, Z_g \ .$$

Both these expressions are dependent only on δ.

Incorrect solutions generated from an incorrect solution. When the partial satisfying assignment is incorrect, at least one of the $\varDelta x$ is nonzero. The expected number of new incorrect satisfying assignments generated from an incorrect satisfying assignment is dependent both on δ and on the behavior of the b_j.

We conjecture the following is close to being true:

Conjecture 1. For random p and q and for $\varDelta x$ not all zero and satisfying

$$\begin{aligned} q\varDelta p + p\varDelta q - \varDelta p \varDelta q &= 0 \pmod{2^i} \\ e\varDelta d + k\varDelta p + k\varDelta q &= 0 \pmod{2^{i+\tau(k)}} \\ e\varDelta d_p - k_p\varDelta p &= 0 \pmod{2^{i+\tau(k_p)}} \\ e\varDelta d_q - k_q\varDelta q &= 0 \pmod{2^{i+\tau(k_q)}} \ , \end{aligned}$$

the next bit of each congruence is 0 or 1 independently with probability near 1/2.

We tested this empirically; each value of the vector (b_1, b_2, b_3, b_4) occurs with probability approximately 1/16. (The error is approximately 5% for $\delta = 0.25$ and $n = 1024$, and approximately 2% for $\delta = 0.25$ and $n = 4096$.)

Let W_b be a random variable denoting the number of bad solutions at bit $i+1$ generated from a single bad solution at bit i. Assuming Conjecture 1,

$$\mathrm{E}\, W_b = \frac{(2-\delta)^5}{16} \qquad \text{and} \qquad \mathrm{E}\, W_b^2 = \mathrm{E}\, W_b + \delta(1-\delta)^4 + 2(1-\delta)^5 \ .$$

Note that the expectation is over the randomness of p and q and the positions of the unknown bits of the key.

When partial knowledge of some of the values (p, q, d, d_p, d_q) is totally unavailable, we can obtain a similar expression.

4.2 Global branching behavior at each step of the program

Now that we have characterized the effect that the constraints have on the branching behavior of the program, we can abstract away all details of RSA entirely and examine the general branching process of the algorithm. We are able to characterize the behavior of the algorithm, and show that if the expected number of branches from any partial solution to the program is less than one, then the total number of branches examined at any step of the program is expected to be constant. All of the following analysis assumes Conjecture 1.

Let X_i be a random variable denoting the number of bad assignments at step i, and recall that Z_g and W_b are random variables denoting the number of bad solutions at bit $i+1$ generated from a single good or bad solution at bit i.

Theorem 2.

$$\mathrm{E}\, X_i = \frac{\mathrm{E}\, Z_g}{1 - \mathrm{E}\, W_b}(1 - (\mathrm{E}\, W_b)^i)$$

This expression can be calculated in a number of ways; we demonstrate how to do so using generating functions in Appendix A.

When $\mathrm{E}\, W_b < 1$, we can bound $\mathrm{E}\, X_i$ from above.

$$\mathrm{E}\, X_i \leq \frac{\mathrm{E}\, Z_g}{1 - \mathrm{E}\, W_b}$$

In the previous section, we calculated expressions for $\mathrm{E}\, Z_g$ and $\mathrm{E}\, W_b$ dependent only on δ, thus when $\mathrm{E}\, W_b < 1$, $\mathrm{E}\, X_i$ can be bounded above by a constant dependent on δ and not on i.

We can evaluate this expression numerically using the values for the expected number of bad solutions discovered in the last section.

In the case with four equations and five unknowns (that is, we have partial knowledge of p, q, d, d_p, and d_q), $\mathrm{E}\, W_b < 1$ at $\delta > 2 - 2^{\frac{4}{5}}$. For $\delta = .2589$, $\mathrm{E}\, X_i < 93247$; for $\delta = .26$, $\mathrm{E}\, X_i < 95$; and for $\delta = .27$ $\mathrm{E}\, X_i < 9$.

In a similar fashion we can obtain the following complicated expression for the variance $\mathrm{Var}\, X_i = \mathrm{E}\, X^2 - (\mathrm{E}\, X)^2$.

Theorem 3.

$$\mathrm{Var}\, X_i = \alpha_1 + \alpha_2(\mathrm{E}\, W_b)^i + \alpha_3(\mathrm{E}\, W_b)^{2i} \tag{14}$$

with

$$\begin{aligned}
\alpha_1 &= \frac{\mathrm{E}\, Z_g \,\mathrm{Var}\, W_b + (1 - \mathrm{E}\, W_b)\,\mathrm{Var}\, Z_g}{(1 - (\mathrm{E}\, W_b)^2)(1 - \mathrm{E}\, W_b)} \\
\alpha_2 &= \frac{\mathrm{E}\, W_b^2 + \mathrm{E}\, W_b - 2\,\mathrm{E}\, W_b\,\mathrm{E}\, Z_g - \mathrm{E}\, Z_g}{1 - \mathrm{E}\, W_b} + 2\left(\frac{\mathrm{E}\, Z_g}{1 - \mathrm{E}\, W_b}\right)^2 \\
\alpha_3 &= -\alpha_1 - \alpha_2 \quad .
\end{aligned}$$

Again evaluating numerically for five unknowns and four equations, at $\delta = .26$ $\mathrm{Var}\, X_i < 7937$, at $\delta = .27$ $\mathrm{Var}\, X_i < 80$, and at $\delta = .28$ $\mathrm{Var}\, X_i < 23$.

4.3 Bounding the total number of keys examined

Now that we have some information about the distribution of the number of partial keys examined at each step, we would like to understand the distribution of the total number of keys examined over an entire run of the program.

We know the expected total number of keys examined for an n-bit key is

$$\mathrm{E}\left[\sum_{i=0}^{n} X_i\right] \leq \frac{\mathrm{E}\, Z_g}{1 - \mathrm{E}\, W_b} n \quad .$$

We will bound how far the total sum is likely to be from this expectation. First, we apply the following bound on the variance of a sum of random variables:

Lemma 2.

$$\operatorname{Var} \sum_{i=1}^{n} X_i \leq n^2 \max_i \operatorname{Var} X_i$$

The proof writes the variance of the sum in terms of covariance, and applies Schwartz's inequality and $\sqrt{ab} \leq \frac{a+b}{2}$.

Apply Chebyshev's inequality to bound the likelihood that $\sum X_i$ is too large:

$$\Pr(|\textstyle\sum_i X_i - \mathrm{E} \sum_i X_i| \geq n\alpha) \leq \dfrac{1}{(n\alpha)^2} \operatorname{Var} \textstyle\sum_i X_i \quad .$$

Apply the above lemma to obtain

$$\Pr(|\textstyle\sum_i X_i - \mathrm{E} \sum_i X_i| \geq n\alpha) \leq \dfrac{1}{\alpha^2} \max\limits_i \operatorname{Var} X_i \quad .$$

When $\delta = .27$, setting $\alpha > 9n$ gives that, for an n-bit key, the algorithm will examine more than $9n^2 + 71n$ potential keys with probability less than $\frac{1}{n^2}$.

4.4 Missing key fields

The same results apply when we have partial knowledge of fewer key fields.

- If the algorithm has partial knowledge of d, p, and q but no information on d_p and d_q, we know that

$$\mathrm{E}\, Z_g = \delta(1-\delta)^2 + (1-\delta)^3 \qquad \mathrm{E}\, Z_g^2 = \mathrm{E}\, Z_g$$
$$\mathrm{E}\, W_b = \frac{(2-\delta)^3}{4} \qquad \mathrm{E}\, W_b^2 = \mathrm{E}\, W_b + \delta(1-\delta)^2 + 2(1-\delta)^3 \quad ,$$

  so $\mathrm{E}\, W_b < 1$ when $\delta > 2 - 2^{\frac{3}{4}} \approx .4126$. Then for $\delta = .42$ the probability that the algorithm examines more than $22n^2 + 24n$ keys is less than $\frac{1}{n^2}$.
- If the algorithm has partial knowledge of p and q but no information on the other values,

$$\mathrm{E}\, Z_g = (1-\delta)^2 \qquad \mathrm{E}\, Z_g^2 = \mathrm{E}\, Z_g$$
$$\mathrm{E}\, W_b = \frac{(2-\delta)^2}{2} \qquad \mathrm{E}\, W_b^2 = \mathrm{E}\, W_b + 2(1-\delta)^2 \quad .$$

  Then $\mathrm{E}\, W_b < 1$ when $\delta > 2 - 2^{\frac{1}{2}} \approx .5859$. When $\delta = .59$ the probability that the algorithm examines more than $29n^2 + 29n$ keys is less than $\frac{1}{n^2}$.

	$n = 512$	768	1024	1536	2048	3072	4096	6144	8192
$\delta = 0.27$	0	0	0	0	0	0	0	0	1
0.26	0	0	0	0	1	5	3	4	8
0.25	0	0	3	6	8	10	17	35	37
0.24	4	5	7	27	50	93	121	201	274

Table 1. Runs (out of 10,000) in which width exceeded 1,000,000

5 Implementation and Performance

We have developed an implementation of our algorithm in approximately 850 lines of C++, using NTL version 5.4.2 and GMP version 4.2.2. Our tests were run, in 64-bit mode, on an Intel Core 2 Duo processor at 2.4 GHz with 4 MB of L2 cache and 4 GB of DDR2 SDRAM at 667 MHz on an 800 MHz bus.

We ran experiments for key sizes between 512 bits and 8192 bits, and for δ values between 0.40 and 0.24. The public exponent is always set to 65537. In each experiment, a key of the appropriate size is randomly censored so that exactly a δ fraction of the bits of the private key components considered together is available to be used for reconstruction. To reduce the time spent on key generation, we reused keys: We generated 100 keys for each key size. For every δ and keysize, we ran 100 experiments with each one of the pregenerated keys, for a total of 10,000 experimental runs. In all, we conducted over 1.1 million runs.

For each run, we recorded the *length* and *width*. The length is the total number of keys considered in the run of the algorithm, at all bit indices; the width is the maximum number of keys considered at any single bit index. These correspond essentially to $\sum_{i=1}^{n/2} X_i$ and $\max_i X_i$, in the notation of Section 4, but can be somewhat larger because we run the algorithm twice in parallel to account for both possible matchings of solutions of (10) to k_p and k_q. To avoid thrashing, we killed runs as soon as the width for some index i exceeded 1,000,000.

When the panic width was not exceeded, the algorithm always ran to completion and correctly recovered the factorization of the modulus.

Of the 900,000 runs of our algorithm with $\delta \geq 0.27$, only a single run ($n = 8192$, $\delta = 0.27$) exceeded the panic width. Applying a Chebyshev bound in this case (with $\operatorname{E} X_i = 9$ and $\operatorname{Var} X_i = 80$) suggests that a width of 1,000,000 should happen with extremely low probability.

Even below $\delta = 0.27$, our algorithm almost always finished within the allotted time. Table 1 shows the number of runs (out of 10,000) in which the panic width was exceeded for various parameter settings. Even for $n = 8192$ and $\delta = 0.24$, our algorithm recovered the factorization of the modulus in more than 97% of all runs. And in many of the overly long runs, the number of bits recovered before the panic width was exceeded suffices to allow recovering the rest using the lattice methods considered in Section 2; this is true of 144 of the 274 very long runs at $n = 8192$ and $\delta = 0.24$, for example.

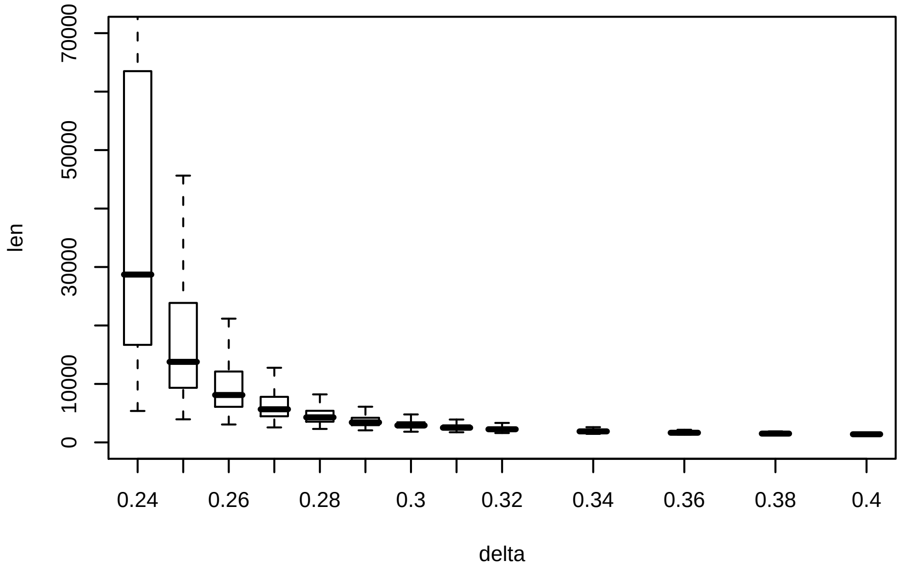

Fig. 1. Boxplot for total number of keys examined by algorithm for $n = 2048$, varying δ.

As expected, search runtime was essentially linear in the total number of keys examined. For $n = 1024$, for example, examining a single key took approximately 5 μsec; for $n = 6144$, approximately 8 μsec. The setup time varied depending on whether k was closer to 0 or to e, but never exceeded 210 msec, even for $n = 8192$.

The plot in Figure 1 gives the behavior for $n = 2048$. For each value of δ we show, using a boxplot, the distribution of the total number of keys examined by runs of the algorithm – i.e., the length of the run. (In our boxplot, generated using R's `boxplot` function, the central bar corresponds to the median, the hinges to the first and third quartiles, and the whisker extents depend on the interquartile range.)

In the full version of this paper [11] we undertake additional analysis of the runtime data.

Acknowledgments

We thank Dan Boneh for suggesting the connection to Hensel lifting; Amir Dembo for improving our branching process analysis; Daniele Micciancio for extensive discussions on using lattice reduction to solve the knapsack problem implicit in our attack; and Eric Rescorla for his help with analyzing the observed runtimes of our algorithm.

In addition, we had fruitful discussions with J. Alex Halderman, Howard Karloff, and N. J. A. Sloane. We would also like to thank the anonymous Crypto reviewers for their comments and suggestions.

This material is based in part upon work supported by the National Science Foundation under CNS grant no. 0831532 (Cyber Trust) and a Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In O. Reingold, editor, *Proceedings of TCC 2009*, volume 5444 of *LNCS*, pages 474–95. Springer-Verlag, Mar. 2009.
2. J. Alwen, Y. Dodis, and D. Wichs. Public key cryptography in the bounded retrieval model and security against side-channel attacks. In S. Halevi, editor, *Proceedings of Crypto 2009*, LNCS. Springer-Verlag, Aug. 2009. This volume.
3. D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2):203–13, Feb. 1999.
4. D. Boneh, G. Durfee, and Y. Frankel. An attack on RSA given a small fraction of the private key bits. In K. Ohta and D. Pei, editors, *Proceedings of Asiacrypt 1998*, volume 1514 of *LNCS*, pages 25–34. Springer-Verlag, Oct. 1998.
5. D. Boneh and H. Shacham. Fast variants of RSA. *RSA Cryptobytes*, 5(1):1–9, Winter/Spring 2002.
6. D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptology*, 10(4):233–60, Dec. 1997.
7. J.-S. Coron and A. May. Deterministic polynomial-time equivalence of computing the RSA secret key and factoring. *J. Cryptology*, 20(1):39–50, Jan. 2007.
8. Y. Dodis, Y. Tauman Kalai, and S. Lovett. On cryptography with auxiliary input. In M. Mitzenmacher, editor, *Proceedings of STOC 2009*. ACM Press, May 2009.
9. S. Goldwasser. Cryptography without (hardly any) secrets? In A. Joux, editor, *Proceedings of Eurocrypt 2009*, volume 5479 of *LNCS*, pages 369–70. Springer-Verlag, Apr. 2009. Invited talk.
10. J. A. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: Cold boot attacks on encryption keys. In P. Van Oorschot, editor, *Proceedings of USENIX Security 2008*, pages 45–60. USENIX, July 2008.
11. N. Heninger and H. Shacham. Reconstructing RSA private keys from random key bits. Cryptology ePrint Archive, Report 2008/510, Dec. 2008. `http://eprint.iacr.org/`. Full version of this paper.
12. M. Herrmann and A. May. Solving linear equations modulo divisors: On factoring given any bits. In J. Pieprzyk, editor, *Proceedings of Asiacrypt 2008*, volume 5350 of *LNCS*, pages 406–24. Springer-Verlag, Dec. 2008.
13. S. Karlin and H. M. Taylor. *A First Course in Stochastic Processes.* Academic Press, 1975.
14. U. Maurer. On the oracle complexity of factoring integers. *Computational Complexity*, 5(3/4):237–47, Sept. 1995.
15. A. May. *New RSA Vulnerabilities Using Lattice Reduction Methods.* PhD thesis, University of Paderborn, Oct. 2003.
16. A. May. Using LLL-reduction for solving RSA and factorization problems: A survey. In P. Nguyen, editor, *Proceedings of LLL+25*, June 2007.

17. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
18. M. Naor and G. Segev. Public-key cryptosystems resilient to key leakage. In S. Halevi, editor, *Proceedings of Crypto 2009*, LNCS. Springer-Verlag, Aug. 2009. This volume.
19. P. Nguyen and J. Stern. Adapting density attacks to low-weight knapsacks. In B. Roy, editor, *Proceedings of Asiacrypt 2005*, volume 3788 of *LNCS*, pages 41–58. Springer-Verlag, Dec. 2005.
20. K. Pietrzak. A leakage-resilient mode of operation. In A. Joux, editor, *Proceedings of Eurocrypt 2009*, volume 5479 of *LNCS*, pages 462–82. Springer-Verlag, Apr. 2009.
21. R. Rivest and A. Shamir. Efficient factoring based on partial information. In F. Pichler, editor, *Proceedings of Eurocrypt 1985*, volume 219 of *LNCS*, pages 31–4. Springer-Verlag, Apr. 1985.
22. RSA Laboratories. PKCS #1 v2.1: RSA cryptography standard, June 2002. Online: `http://www.rsa.com/rsalabs/node.asp?id=2125`.
23. S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: Results from the 2008 Debian OpenSSL debacle, May 2009. Manuscript.

A Computing the Expectation and Variance

In this appendix, we derive expressions for the expectation and variance of the number of incorrect keys generated at each step of the program. Let X_i be a random variable denoting the number of bad assignments at step i. We will calculate the expectation $\mathrm{E}\,X_i$ and variance $\mathrm{Var}\,X_i$. (We know that the number of good assignments is always equal to one.)

To calculate these values, we will use probability generating functions. For more information on this approach, see e.g., [13, Ch. 8]. A probability generating function $F(s) = \sum \Pr[X = k]s^k$ represents the distribution of the discrete random variable X. $F(s)$ satisfies the following identities:

$$F(1) = 1 \ , \qquad \mathrm{E}\,X = F'(1) \ , \qquad \text{and} \qquad \mathrm{Var}\,X = F''(1) + F'(1) - F'(1)^2 \ .$$

Let $G_i(s)$ be the probability generating function for the X_i, $z(s)$ the probability generating function for the Z_g (the number of bad assignments generated from a correct assignment) and $w(s)$ the probability generating function for the W_b (the number of bad assignments generated from a bad assignment).

From Section 4, we know that

$$z'(1) = \mathrm{E}\,Z_g \ , \qquad z''(1) = \mathrm{E}\,Z_g^2 - \mathrm{E}\,Z_g \ ,$$
$$w'(1) = \mathrm{E}\,W_b \ , \qquad \text{and} \qquad w''(1) = \mathrm{E}\,W_b^2 - \mathrm{E}\,W_b \ .$$

Expectation of X_i. We will calculate $\mathrm{E}\,X_i = G_i'(1)$. $G_i(s)$ satisfies the recurrence

$$G_{i+1}(s) = G_i(w(s))z(s) \ , \tag{15}$$

that is, that the number of bad solutions at each step is equal to the number of bad solutions lifted from bad solutions plus the number of bad solutions produced from good solutions. (Recall that a generating function for the sum of two independent random variables is given by the convolution of their generating functions.) We also have that

$$G_0(s) = 1 \ ,$$

because initially there are no bad solutions. Differentiating (15) gives

$$G_i'(s) = (G_{i-1}(w(s))w'(s)z(s) + G_{i-1}(w(s))z'(s) \ . \tag{16}$$

Set $s = 1$ and use the fact that $G_i(1) = w(1) = z(1) = 1$ to obtain

$$G_i'(1) = w'(1)G_{i-1}'(1) + z'(1) \ .$$

Solving the recurrence yields

$$G_i'(1) = \frac{z'(1)}{1 - w'(1)}(1 - (w'(1))^i) \ . \tag{17}$$

If $w'(1) < 1$, then $w'(1)^i$ tends to 0 as i increases and

$$EX_i = G_i'(1) < \frac{z'(1)}{1 - w'(1)} \tag{18}$$

for all i. The expected number of bad solutions at any step of the process will be bounded by a value dependent only on δ and not on i.

Variance of X_i. To compute the variance $\operatorname{Var} X_i = G_i''(1) + G_i'(1) - (G_i'(1))^2$, we differentiate (16) again to obtain

$$\begin{aligned} G_i''(s) = G_{i-1}''(w(s))w'(s)w'(s)z(s) + G_{i-1}'(w(s))w''(s)z(s) \\ + 2G_{i-1}'(w(s))w'(s)z'(s) + G_{i-1}(w(s))z''(s) \ . \end{aligned} \tag{19}$$

Evaluating at $s = 1$ gives

$$G_i''(1) = G_{i-1}''(1)w'(1)^2 + G_{i-1}'(1)w''(1) + 2G_{i-1}'(1)w'(1)z'(1) + z''(1) \ .$$

Substitute in (17) to get

$$\begin{aligned} G_i''(1) = G_{i-1}''(1)w'(1)^2 + \frac{z'(1)}{1 - w'(1)}(1 - (w'(1))^i)w''(1) \\ + 2\frac{z'(1)}{1 - w'(1)}(1 - (w'(1))^i)w'(1)z'(1) + z''(1) \ . \end{aligned} \tag{20}$$

The general solution to this recurrence is

$$G_i''(1) = c_1 + c_2 w'(1)^i + c_3 w'(1)^{2i} \tag{21}$$

with

$$\begin{aligned} c_1 &= \frac{1}{1 - w'(1)^2}\left(\frac{z'(1)}{1 - w'(1)}(w''(1) + 2w'(1)z'(1)) + z''(1)\right) \\ c_2 &= -\frac{1}{1 - w'(1)}(w''(1) + 2w'(1)z'(1)) \\ c_3 &= -c_1 - c_2 \ . \end{aligned}$$