A shorter version of this paper will appear in *Proc. 21st USENIX Security Symposium*, Aug. 2012. Rev. 2; July 11, 2012.
For the newest revision of this paper, partial source code, and our online key-check service, visit **https://factorable.net**.

Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices

Nadia Heninger[†][*] Zakir Durumeric[‡][*] Eric Wustrow[‡] J. Alex Halderman[‡]

[†] *University of California, San Diego*
nadiah@cs.ucsd.edu

[‡] *The University of Michigan*
{zakir, ewust, jhalderm}@umich.edu

Abstract

RSA and DSA can fail catastrophically when used with malfunctioning random number generators, but the extent to which these problems arise in practice has never been comprehensively studied at Internet scale. We perform the largest ever network survey of TLS and SSH servers and present evidence that vulnerable keys are surprisingly widespread. We find that 0.75% of TLS certificates share keys due to insufficient entropy during key generation, and we suspect that another 1.70% come from the same faulty implementations and may be susceptible to compromise. Even more alarmingly, we are able to obtain RSA private keys for 0.50% of TLS hosts and 0.03% of SSH hosts, because their public keys shared nontrivial common factors due to entropy problems, and DSA private keys for 1.03% of SSH hosts, because of insufficient signature randomness. We cluster and investigate the vulnerable hosts, finding that the vast majority appear to be headless or embedded devices. In experiments with three software components commonly used by these devices, we are able to reproduce the vulnerabilities and identify specific software behaviors that induce them, including a boot-time entropy hole in the Linux random number generator. Finally, we suggest defenses and draw lessons for developers, users, and the security community.

1 Introduction and Roadmap

Randomness is essential for modern cryptography, where security often depends on keys being chosen uniformly at random. Researchers have long studied random number generation, from both practical and theoretical perspectives (e.g., [9, 14, 17, 19, 24, 26]), and a handful of major vulnerabilities (e.g., [6, 22]) have attracted considerable scrutiny to some of the most critical implementations. Given the importance of this problem and the effort and attention spent improving the state of the art, one might expect that today's widely used operating systems and server software generate random numbers securely. In this paper, we test that proposition empirically by examining the public keys in use on the Internet.

The first component of our study is the most comprehensive Internet-wide survey to date of two of the most important cryptographic protocols, TLS and SSH (Section 3.1). By scanning the public IPv4 address space, we collected 5.8 million unique TLS certificates from 12.8 million hosts and 6.2 million unique SSH host keys from 10.2 million hosts. This is 67% more TLS hosts than the latest released EFF SSL Observatory dataset [20]. Our techniques take less than 24 hours to scan the entire address space for listening hosts and less than 96 hours to retrieve keys from them. The results give us a macroscopic perspective of the universe of keys.

Next, we analyze this dataset to find evidence of several kinds of problems related to inadequate randomness. To our surprise, at least 5.57% of TLS hosts and 9.60% of SSH hosts use the same keys as other hosts in an apparently vulnerable manner (Section 4.1). In the case of TLS, at least 5.23% of hosts use manufacturer default keys that were never changed by the owner, and another 0.34% appear to have generated the same keys as one or more other hosts due to malfunctioning random number generators. Only a handful of the vulnerable TLS certificates are signed by browser-trusted certificate authorities.

Even more alarmingly, we are able to compute the private keys for 64,000 (0.50%) of the TLS hosts and 108,000 (1.06%) of the SSH hosts from our scan data alone by exploiting known weaknesses of RSA and DSA when used with insufficient randomness. In the case of RSA, distinct moduli that share exactly one prime factor will result in public keys that appear distinct but whose private keys are efficiently computable by calculating the greatest common divisor (GCD). We implemented an algorithm that can compute the GCDs of all pairs of 11 million distinct public RSA moduli in less than 2 hours (Section 3.3). Using the resulting factors, we are able to

[*]The first two authors both made substantial contributions.

obtain the private keys for 0.50% of TLS hosts and 0.03% of SSH hosts (Section 4.2). In the case of DSA, if a DSA key is used to sign two different messages with the same ephemeral key, an attacker can efficiently compute the signer's long-term private key. We find that our SSH scan data contain numerous DSA signatures that used the same ephemeral keys during signing, allowing us to compute the private keys for 1.6% of SSH DSA hosts (Section 4.3).

To understand why these problem are occurring, we manually investigated hundreds of the vulnerable hosts, which were representative of the most commonly repeated keys as well as each of the private keys we obtained (Section 3.2). Nearly all served information identifying them as headless or embedded systems, including routers, server management cards, firewalls, and other network devices. Such devices typically generate keys automatically on first boot, and may have limited entropy sources compared to traditional PCs. Furthermore, when we examined clusters of hosts that shared a key or factor, in nearly all cases these appeared to be linked by a manufacturer or device model. These observations lead us to conclude that the problems are caused by specific defective implementations that generate keys without having collected sufficient entropy. We identified vulnerable devices and software from 54 manufacturers, including some of the largest names in the technology industry, and worked to notify the responsible parties.

In the final component of our study, we experimentally explore the root causes of these vulnerabilities by investigating several of the most common open-source software components from the population of vulnerable devices (Section 5). Based on the devices we identified, it is clear that no one implementation is solely responsible, but we are able to reproduce the vulnerabilities in plausible software configurations. Every software package we examined relies on `/dev/urandom` to generate cryptographic keys; however, we find that Linux's random number generator (RNG) can exhibit a boot-time entropy hole that causes `urandom` to produce deterministic output under conditions likely to occur in headless and embedded devices. In experiments with OpenSSL and Dropbear SSH, we show how repeated output from the system RNG can lead not only to repeated long-term keys but also to factorable RSA keys and repeated DSA ephemeral keys due to the behavior of application-specific entropy pools.

Given the diversity of the devices and software implementations involved, mitigating these problems will require action by many different parties. We draw lessons and recommendations for developers of operating systems, cryptographic libraries, and applications, and for device manufacturers, certificate authorities, end users, and the security and cryptography communities (Section 7). We have also created an online key-check service to allow users to test whether their keys are vulnerable (Section 8).

It is natural to wonder whether these results should call into question the security of every RSA or DSA key. Based on our analysis, the margin of safety is slimmer than we might like, but we have no reason to doubt the security of most keys generated interactively by users on traditional PCs. While we took advantage of the details of specific cryptographic algorithms in this paper, we conclude that the blame for these vulnerabilities lies chiefly with the implementations. Ultimately, the results of our study should serve as a wake-up call that secure random number generation continues to be an unsolved problem in important areas of practice.

Online resources For the most recent version of this paper, partial source code, updated advisories, and our online key-check service, visit https://factorable.net.

2 Background

In this section, we review the RSA and DSA public-key cryptosystems and discuss the known weaknesses of each that we used to compromise private keys. We then discuss how an adversary might exploit compromised keys to attack SSH and TLS in practice.

2.1 RSA review

An RSA [45] public key consists of two integers: an exponent e and a modulus N. The exponent can be shared among multiple public keys without compromising security, but the modulus cannot. The modulus N is the product of two randomly chosen prime numbers p and q. The private key is the decryption exponent

$$d = e^{-1} \bmod (p-1)(q-1).$$

Anyone who knows the factorization of N can efficiently compute the private key for any public key (e,N) using the preceding equation. When p and q are unknown, the most efficient known method to calculate the private key is to factor N into p and q and use the above equation to calculate d [10].

Factorable RSA keys No one has been publicly known to factor a well-generated 1024-bit RSA modulus; the largest known factored modulus is 768 bits, which was announced in December 2009 after a multi-year distributed-computing effort [34]. In contrast, the greatest common divisor (GCD) of two 1024-bit integers can be computed in microseconds. This asymmetry leads to a well-known vulnerability: if an attacker can find two distinct RSA moduli N_1 and N_2 that share a prime factor p but have different second prime factors q_1 and q_2, then the attacker can easily factor both moduli by computing their GCD, p, and dividing to find q_1 and q_2. The attacker can then compute both private keys as explained above.

2.2 DSA review

A DSA [38] public key consists of three so-called domain parameters (two prime moduli p and q and a generator g of the subgroup of order q mod p) and an integer $y = g^x \bmod p$, where x is the private key. The domain parameters may be shared among multiple public keys without compromising security. A DSA signature consists of a pair of integers (r,s): $r = g^k \bmod p \bmod q$ and $s = (k^{-1}(H(m)+xr)) \bmod q$, where k is a randomly chosen ephemeral private key and $H(m)$ is the hash of the message.

Low-entropy DSA signatures DSA is known to fail catastrophically if the ephemeral key k used in the signing operation is generated with insufficient entropy [5]. (Elliptic curve DSA (ECDSA) is similarly vulnerable. [12]) If k is known for a signature (r,s), then the private key x can be computed from the signature and public key as follows:

$$x = r^{-1}(ks - H(m)) \bmod q.$$

If a DSA private key is used to sign two different messages with the same k, then an attacker can efficiently compute the value k from the public key and signatures and use the above equation to compute the private key x [35]. If two messages m_1 and m_2 were signed using the same ephemeral key k to obtain signatures (r_1,s_1) and (r_2,s_2), then this will be immediately clear as r_1 and r_2 will be equal. The ephemeral key k can be computed as:

$$k = (H(m_1) - H(m_2))(s_1 - s_2)^{-1} \bmod q.$$

2.3 Attack scenarios

The weak key vulnerabilities we describe in this paper can be exploited to compromise two of the most important cryptographic transport protocols used on the Internet, TLS and SSH, both of which commonly use RSA or DSA to authenticate servers to clients.

TLS In TLS [18], the server sends its public key in a TLS certificate during the protocol handshake. The key is used either to provide a signature on the handshake (when Diffie-Hellman key exchange is negotiated) or to encrypt session key material chosen by the client (when RSA-encrypted key exchange is negotiated).

If the key exchange is RSA encrypted, a passive eavesdropper with the server's private key can decrypt the message containing the session key material and use it to decrypt the entire session. If the session key is negotiated using Diffie-Hellman key exchange, then a passive attacker will be unable to compromise the session key from just a connection transcript. However, in both cases, an active attacker who can intercept and modify traffic between the client and server can man-in-the-middle the connection in order to decrypt or modify the traffic.

SSH In SSH, host keys allow a server to authenticate itself to a client by providing a signature during the protocol handshake. There are two major versions of the protocol. In SSH-1 [52], the client encrypts session key material using the server's public key. SSH-2 [53] uses a Diffie-Hellman key exchange to establish a session key. The user manually verifies the host key fingerprint the first time she connects to an SSH server. Most clients then store the key locally in a `known_hosts` file and automatically trust it for all subsequent connections.

As in TLS, a passive eavesdropper with a server's private key can decrypt an entire SSH-1 session. However, because SSH-2 uses Diffie-Hellman, it is vulnerable only to an active man-in-the-middle attack. In the SSH user authentication protocol, the user-supplied password is sent in plaintext over the encrypted channel. An attacker who knows a server's private key can use the above attacks to learn a user's password and escalate an attack to the system.

3 Methodology

In this section, we explain how we performed our Internet-wide survey of public keys, how we attributed vulnerable keys to devices, and how we efficiently factored poorly generated RSA keys.

3.1 Internet-wide scanning

We performed our data collection in three phases: discovering IP addresses accepting connections on TCP port 443 (HTTPS) or 22 (SSH); performing a TLS or SSH handshake and storing the presented certificate chain or host key; and parsing the collected certificates and host keys into a relational database. Table 1 summarizes the results.

Host discovery In the first phase, we scanned the public IPv4 address space to find hosts with port 443 or 22 open. We used the Nmap 5 network exploration tool [39] to perform a SYN scan[1], which involves sending a TCP SYN packet to each candidate host and detecting whether the host responds with a SYN-ACK packet. We chose this scanning method based on its low bandwidth requirements; for the vast majority of hosts (those with the target port closed), at most two packets need to be exchanged. We excluded address ranges reserved for location identification, private use, loopback, link local, multicast, or future use in the IANA IPv4 address space registry [32].

[1]The Nmap options we used were: `-sS -Pn -n -T5 --min-hostgroup=2000 --max-rtt-timeout=500ms --min-rate=10000` We selected these parameters by choosing the most aggressive parameters that did not cause non-negligible dropoff in response on an EC2 Micro Instance.

	SSL Observatory (12/2010)	Our TLS scan (10/2011)	Our SSH scans (2-4/2012)
Hosts with open port 443 or 22	≈16,200,000	28,923,800	23,237,081
Completed protocol handshakes	7,704,837	12,828,613	10,216,363
Distinct RSA public keys	3,933,366	5,656,519	3,821,639
Distinct DSA public keys	1,906	6,241	2,789,662
Distinct TLS certificates	4,021,766	5,847,957	—
Trusted by major browsers	1,455,391	1,956,267	—

Table 1: **Internet-wide scan results** — We exhaustively scanned the public IPv4 address space for TLS and SSH servers listening on ports 443 and 22, respectively. Our results constitute the largest such network survey reported to date. For comparison, we also show statistics for the EFF SSL Observatory's most recent public dataset [20].

We executed our first host discovery scan beginning on October 6, 2011 from 25 Amazon EC2 Micro instances spread across five EC2 regions (Virginia, California, Japan, Singapore, and Ireland). The scan ran at an average of 40,566 IPs/second and finished in 25 hours; it found 28,923,800 IPs with port 443 open. We performed a second scan to find SSH hosts, beginning on February 12, 2012; it found 23,237,081 IPs with port 22 open.

Certificate and host-key retrieval In the second phase of the scanning process, we attempted a TLS or SSH protocol handshake with the addresses accepting connections on port 443 or 22 and recorded the TLS certificate or SSH host key presented by the server.

For TLS, we implemented a certificate fetcher in Python using the Twisted event-driven network framework [33]. We fetched TLS certificates using an EC2 Large instance with five processes each maintaining 800 concurrent connections. We started fetching certificates on October 11, 2011. The fetcher processed a mean of 83 hosts/second and completed in approximately 96 hours. It retrieved certificate chains from 12,828,613 IP addresses (44.4% of hosts listening on port 443). These certificate chains contained 5,656,519 distinct RSA public keys and 6,241 distinct DSA public keys.

To efficiently collect SSH host keys, we implemented a simple SSH client in C, which is able to process upwards of 1200 hosts/second by concurrently performing protocol handshakes using *libevent* [40]. Our client initiates a normal-looking connection and offers only `Diffie-Hellman Group 1` key exchange[2], and then downloads and stores the host key provided by the server.

Initially, we ran the fetcher from an EC2 Large instance in a run that started on February 12, 2012. This run targeted only RSA-based host keys and returned 3,821,639 distinct keys from 10,216,363 IP addresses (44.0% of hosts listening on port 22). In two later runs, we targeted DSA-based host keys, and rescanned those hosts that had offered DSA keys in the first SSH scan. For these, we also stored the authentication signature provided by the server; we varied the client string to ensure that each signature would be distinct. The first DSA run started on March 26, 2012 from a host at UCSD; it returned 2,091,643 distinct DSA host keys from 4,572,218 IP addresses (51.4% of listening hosts). The second run, from a host at the University of Michigan, started on April 1, 2012; it took 3 hours to complete and returned 2,058,706 distinct DSA host keys from 4,542,707 IP addresses.

TLS certificate processing For TLS, we performed a third processing stage in which we parsed the previously fetched certificate chains and generated a database from the X.509 fields. We implemented a certificate parser in Python and C primarily based on the M2Crypto [46] SWIG [3] interface to the OpenSSL library [16]. In total, we found 5,847,957 distinct certificates, of which 1,956,267(33.5%) were browser trusted. We deemed a certificate to be trusted if we could find a chain of trust leading to a root CA trusted by the current version of Firefox, MacOS, OpenSSL, or Windows.

3.2 Identifying vulnerable device models

We attempted to determine what hardware and software generated or served the weak keys we identified using manual detective work. The most straightforward method was based on TLS certificate information—predominately the X.509 *subject* and *issuer* fields. In many cases, the certificate identified a specific manufacturer or device model. Other certificates contained less information; we attempted to identify these devices through Nmap host detection or by inspecting the public contents of HTTPS sites or other IP services hosted on the IP addresses.

When we could identify a pattern in vulnerable TLS certificates that appeared to belong to a device model or product line, we constructed regular expressions to find other similar devices in our scan results. Under the theory that the keys were vulnerable because of a problem with

[2]Our SSH client implementation is greatly simplified by only supporting the `Diffie-Hellman Group 1` key exchange, which is supported by all SSH servers.

the design of the devices (where they were most likely generated), this allows us to estimate the total population of devices that might be potentially vulnerable, beyond those serving immediately compromised keys.

Identifying SSH devices was more problematic, as SSH keys do not include descriptive fields and the server identification string used in the protocol often indicated only a common build of a popular SSH server. We were able to classify many of the vulnerable SSH hosts using a combination of TCP/IP fingerprinting and examination of information served over HTTP and HTTPS.

When we were able to identify a vulnerable device model, we attempted to disclose the problem to the manufacturer. We provided these manufacturers with background on the vulnerabilities, selected vulnerable IP addresses from among the hosts we identified, and suggestions for remedying the problem.

The device names and manufacturers that we report here have been identified with moderate or high confidence given the available information. However, because we do not have physical access to the hosts, we cannot state with certainty that all our identifications are correct.

3.3 Efficiently computing all-pairs GCDs

We now describe how we efficiently computed the pairwise GCD of all distinct RSA moduli in our multimillion-key dataset. This allowed us to calculate RSA private keys for 66,540 vulnerable hosts that shared one of their RSA prime factors with another host in our survey.

The fastest known factoring method for general integers is the number field sieve, which has heuristic complexity $O(2^{n^{1/3}(\log n)^{2/3}})$ for n-bit numbers [36]. With current computing power, factoring any of the 85,988 512-bit RSA public keys detected in our scan is within the reach of a dedicated amateur using existing implementations and common hardware [49], but factoring *all of them* would require thousands of years of computation.

In contrast to factoring, the greatest common divisor (GCD) of two integers can be computed very efficiently using Euclid's algorithm, with computational complexity $O(n^2)$ bits for n-bit numbers. Using fast integer arithmetic, the complexity of GCD can be improved to $O(n(\lg n)^2 \lg\lg n)$ [8]. Computing the GCD of two 1024-bit RSA moduli using the GMP library [23] takes approximately 15 μs on a current mid-range computer.

The naïve way to compute the GCDs of every pair of integers in a large set would be to apply a GCD algorithm to each pair individually. There are 6×10^{13} distinct pairs of RSA moduli in our data; at 15 μs per pair, this calculation would take 30 years. We can do much better by using a more efficient algorithm.

To accomplish this, we implemented a quasilinear-time algorithm for factoring a collection of integers into coprimes, due to Bernstein [7]. The relevant steps, illustrated in Figure 1, are as follows:

$$\begin{tikzcd}
N_1 \arrow[d] & N_2 \arrow[dl] & N_3 \arrow[d] & N_4 \arrow[dl] & \\
\times \arrow[dr] & & \times \arrow[dl] & & \text{product tree} \\
& N_1N_2N_3N_4 \arrow[dl] \arrow[dr] & & & \\
\bmod N_1^2N_2^2 \arrow[d] \arrow[dr] & & \bmod N_3^2N_4^2 \arrow[d] \arrow[dr] & & \text{remainder tree} \\
\bmod N_1^2 \arrow[d] & \bmod N_2^2 \arrow[d] & \bmod N_3^2 \arrow[d] & \bmod N_4^2 \arrow[d] & \\
/N_1 \arrow[d] & /N_2 \arrow[d] & /N_3 \arrow[d] & /N_4 \arrow[d] & \\
\gcd(\cdot, N_1) & \gcd(\cdot, N_2) & \gcd(\cdot, N_3) & \gcd(\cdot, N_4) &
\end{tikzcd}$$

Figure 1: **Computing all-pairs GCDs efficiently** — We computed the GCD of every pair of RSA moduli in our dataset using an algorithm due to Bernstein [7]: First, compute the product of all moduli using a product tree, then use a remainder tree to reduce the product modulo each input. The final output is the GCD of each input modulus with the product of all the other moduli.

Algorithm 1 Quasilinear GCD finding

```
Input: $N_1, \ldots, N_m$ RSA moduli
1: Compute $P = \prod N_i$ using a product tree.
2: Compute $z_i = (P \bmod N_i^2)$ for all $i$
   using a remainder tree.
Output: $\gcd(N_i, z_i/N_i)$ for all $i$.
```

A product tree computes the product of m numbers by constructing a binary tree of products. A remainder tree computes the remainder of an integer modulo many integers by successively computing remainders for each node in their product tree. For further discussion, see Bernstein [8].

The final output of the algorithm is the GCD of each modulus with the product of all the other moduli. We are interested in the moduli for which this GCD is not 1. However, if a modulus shares both of its prime factors with two other distinct moduli, then the GCD will be the modulus itself rather than one of its prime factors—therefore providing us no immediate factorization. This occurred in a handful of instances in our dataset; we factored these moduli using the naïve quadratic algorithm for pairwise GCDs.

Using fast integer arithmetic algorithms, multiplication and modular reduction on n-bit integers can be done in time $O(n \lg n \lg\lg n)$, and GCDs in time $n(\lg n)^2 \lg\lg n$ [8]. Thus, the asymptotic running time of Algorithm 1 on m

	Our TLS Scan		Our SSH Scans	
Number of live hosts	12,828,613	(100.00%)	10,216,363	(100.00%)
. . . using repeated keys	7,770,232	(60.50%)	6,642,222	(65.00%)
. . . using vulnerable repeated keys	714,243	(5.57%)	981,166	(9.60%)
. . . using default certificates or default keys	670,391	(5.23%)		
. . . using low-entropy repeated keys	43,852	(0.34%)		
. . . using RSA keys we could factor	64,081	(0.50%)	2,459	(0.03%)
. . . using DSA keys we could compromise			105,728	(1.03%)
. . . using Debian weak keys	4,147	(0.03%)	53,141	(0.52%)
. . . using 512-bit RSA keys	123,038	(0.96%)	8,459	(0.08%)
. . . identified as a vulnerable device model	985,031	(7.68%)	1,070,522	(10.48%)
. . . model using low-entropy repeated keys	314,640	(2.45%)		

Table 2: **Summary of vulnerabilities** — We analyzed our TLS and SSH scan results to measure the population of hosts exhibiting several entropy-related vulnerabilities. These include use of repeated keys, use of RSA keys that were factorable due to repeated primes, and use of DSA keys that were compromised by repeated signature randomness. Under the theory that vulnerable repeated keys were generated by embedded or headless devices with defective designs, we also report the number of hosts that we identified as these device models. Many of these hosts may be at risk even though we did not specifically observe repeats of their keys.

n-bit integers would be $O(mn \lg m \lg(mn) \lg\lg(mn))$ for the product and remainder trees and $O(mn(\lg n)^2 \lg\lg n)$ to compute the GCDs.

We implemented the algorithm in C using the GMP library [23] for the arithmetic operations and ran it on the 11,170,883 distinct RSA moduli from our TLS and SSH datasets and the EFF SSL Observatory [20] dataset. Our product tree had 24 levels, each of which was about 2 GB in size. In addition, GMP allocates a large amount of scratch memory during the calculations—around 30 GB for our dataset. Due to memory limitations, we wrote each level to disk. We found that the product of all of the moduli was too large for GMP's raw-integer I/O format, which cannot process numbers larger than 2^{31} bytes. We patched the library to remove this limitation.

The entire computation finished in 5.5 hours using a single core on a machine with a 3.30 GHz Intel Core i5 processor and 32 GB of RAM. The remainder tree took approximately ten times as long to process as the product tree. Parallelized across sixteen cores on an EC2 Cluster Compute Eight Extra Large Instance with 60.5 GB of RAM and using EBS-backed storage for scratch data, the same computation took 1.3 hours at a cost of about $5.

4 Vulnerabilities

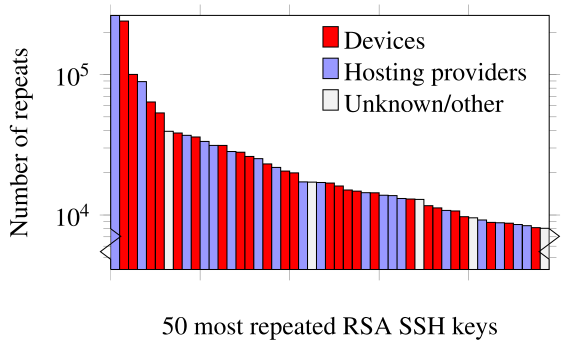
We analyzed the data from our TLS and SSH scans and identified several patterns of vulnerability that would have been difficult to detect without a macroscopic view of the Internet. This section discusses the details of these problems, as summarized in Table 2.

4.1 Repeated keys

We found that 7,770,232 of the TLS hosts (61%) and 6,642,222 of the SSH hosts (65%) served the same key as another host in our scans. To understand why, we clustered certificates and host keys that shared the same public key and manually inspected representatives of the largest clusters. In all but a few cases, the TLS certificate subjects, SSH version strings, or WHOIS information were identical within a cluster, or pointed to a single manufacturer or organization. This sometimes suggested an explanation for the shared keys.

Not all of the repeated keys were due to vulnerabilities. For instance, many of the most commonly repeated keys appeared in shared hosting situations. Six of the ten most common DSA host keys and three of the ten most common RSA host keys were served by large hosting providers (see Figure 2). Another frequent reason for repeated keys was distinct TLS certificates all belonging to the same organization. For example, TLS hosts at google.com, appspot.com, and doubleclick.net all served distinct certificates with the same public key. We excluded these cases and attributed remaining clusters of shared keys to several classes of problems.

Default keys A common reason for hosts to share the same key that we *do* consider a vulnerability is manufacturer-default keys. These are preconfigured in the firmware of many devices, such that every device of a given model shares the same key pair unless the user changes it. The private keys to these devices may be accessible through reverse engineering, and published databases of default keys such as littleblackbox [28] contain private keys for thousands of firmware releases.



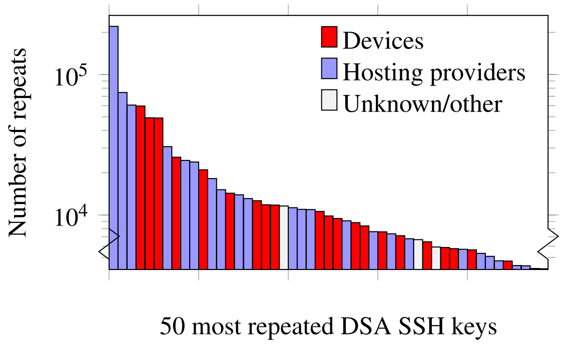


Figure 2: **Commonly repeated SSH keys** — We investigated the 50 most repeated SSH host keys for both RSA and DSA. Nearly all of the repeats appeared to be due either to hosting providers using a single key on many IP addresses or to devices that used a default key or generated keys using insufficient entropy. Note log scale.

At least 670,391 (5.23%) of the TLS hosts appeared to serve manufacturer-default certificates or keys. We identified 57 distinct TLS certificates from the littleblackbox 0.1.3 database, revealing private keys for 23,476 (0.18%) of the TLS hosts. We were able to identify 170 additional default certificates by manually inspecting the 654 self-signed certificates that appeared on more than 250 IP addresses. (Most browser-trusted certificates that were served on multiple IP addresses appeared to be legitimate HTTPS sites served from multiple web servers.) We classified a certificate as a manufacturer default if nearly all the devices of a given model used identical certificates, or if the certificate was labeled as a default certificate. Our manually identified manufacturer-default certificates were served by another 618,014 (4.82%) of the TLS hosts.

The most common default certificate that we could ascribe to a particular device belonged to a model of consumer router. Our scan uncovered 90,779 instances of this device model sharing a single certificate. We also found a variety of enterprise products serving default keys, including server management devices, network storage devices, routers, remote access devices, and VoIP devices.

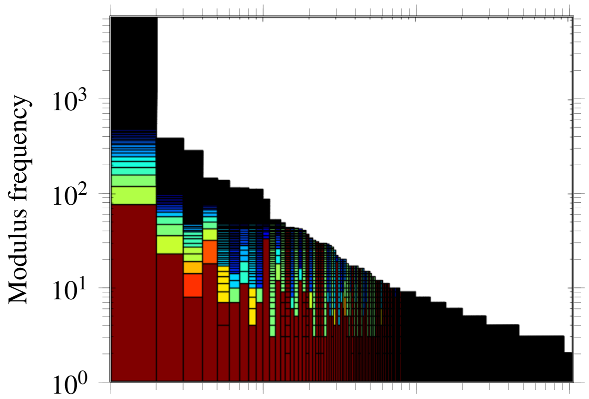
At least another 85,046 TLS hosts (0.66%) served default Apache certificates (sometimes referred to as snake-oil certificates, because they often include the CN `www.snakeoil.dom`); we identified 832 distinct snake-oil certificates that were served on more than one host, as many Linux distributions ship distinct default certificates. We did not count these among the vulnerable manufacturer-default keys. However, we also found 58 non-default certificates that shared an RSA modulus with one of these snake-oil certificates, including 22 that had been signed by browser-trusted CAs. In these cases, it is possible that organizations generated certificates with their own information but included the default public key shipped with Apache into their own certificate, rather than generating a new private key.

We observed a similar phenomenon with default TLS keys served by Citrix remote access servers: we found 1584 apparently unrelated Citrix hosts serving distinct certificates (including 16 signed by trusted CAs) that shared keys with default certificates. The signed certificates belonged to entities including Fortune 500 companies, insurance providers, law firms, a major public transit authority, and the U.S. Navy. These organizations may be authenticating or encrypting remote-access service communication using these duplicate keys.

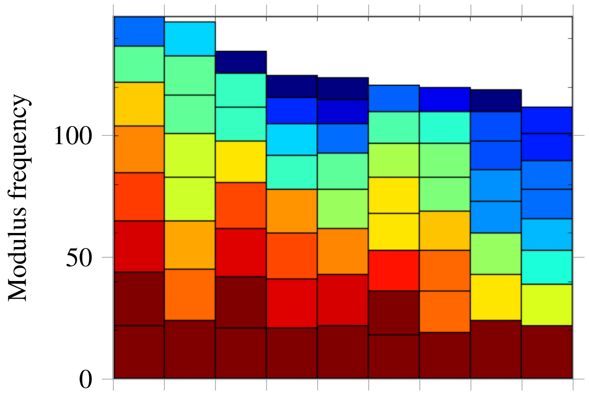
For most of the repeated SSH keys, the lack of uniquely identifying host information prevents us from distinguishing default keys from keys generated with insufficient entropy, so we address these together in the next section.

Repeated keys due to low entropy Another common reason that hosts share the same key appears to be entropy problems during key generation. In these instances, when we investigated a key cluster, we would typically see thousands of hosts across many address ranges, and, when we checked the keys corresponding to other instances of the same model of device, we would see a long-tailed distribution in their frequencies. Intuitively, this type of distribution suggests that the key generation process may be using insufficient entropy, with distinct keys due to relatively rare events. For TLS, our investigations began with the keys that occurred in at least 100 distinct self-signed certificates. For SSH, we started from the 50 most commonly repeated keys for each of RSA (appearing on more than 8000 hosts) and DSA (appearing on more than 4000 hosts).

With this process, we identified 43,852 TLS hosts (0.34%) that served repeated keys due apparently to low entropy during key generation. The repeated keys included 208 distinct TLS RSA keys. 27,545 certificates (98%) containing these repeated keys were self-signed; all 577 CA-signed certificates identified Iomega StorCenter network storage devices. For most SSH hosts we were un-



(a) Primes generated by Juniper network security devices



(b) Primes generated by IBM remote management cards

Figure 3: **Visualizing RSA common factors** — Different implementations displayed different patterns of vulnerable keys. These plots depict the distribution of vulnerable keys divisible by common factors generated by two different device models. Each column represents a collection of RSA moduli divisible by a single prime factor p. The color and internal rectangles show, for each p, the frequencies of each distinct prime factor q. The Juniper device (*left*; note log-log scale) follows a long-tailed distribution that is typical of many of the devices we identified. In contrast, the IBM remote access device (*right*) was unique among those we observed in that it generates RSA moduli roughly uniformly distributed among nine distinct prime factors.

able to distinguish between default keys and keys repeated due to entropy problems, but 981,166 of the SSH hosts (9.60%) served keys repeated for one of these reasons. Our TLS and SSH datasets contained two RSA keys in common; one was served from the same host on both TLS and SSH, and the other was served by TLS and SSH on distinct and seemingly unrelated hosts.

We used the techniques described in Section 3.2 to identify apparently vulnerable devices from 27 manufacturers. These include enterprise-grade routers from Cisco; server management cards from Dell, Hewlett-Packard, and IBM; VPN devices; building security systems; network attached storage devices; and several kinds of consumer routers and VoIP products.

Given cryptographic key sizes, we would not expect to see devices generate a single duplicated key for the population sizes we examined if the keys were generated with sufficient entropy. Therefore, duplicated keys are a red flag that calls the security of the device's key generation process into question, and all keys generated with the same model device should be considered suspect. For 14 of the TLS devices generating repeated keys, we were able to infer a fingerprint for the device model and estimate the total population of the device in our scan. The prevalence of duplicated keys varied greatly for different device models, from as low as 0.2% in the case of one router to 98% for one thin client. The total population of these identified, potentially vulnerable TLS devices was 314,640 hosts, which represents 2.45% of the TLS hosts in our scan.

In the above analyses, we excluded repeated keys that were due to the infamous Debian weak-key vulnerability [6, 51]. 4,147 (0.03%) of the hosts in our TLS dataset served certificates with Debian weak keys. 17 of the 2,904 vulnerable certificates were signed by browser-trusted authorities and 11 were signed after the vulnerability was announced in August 2008. 31,111 (0.34%) of the RSA SSH hosts and 22,030 (0.34%) of the DSA SSH hosts served Debian weak keys. The most commonly repeated weak key appeared on 3,422 SSH RSA hosts and 2,357 SSH DSA hosts. The most repeated Debian weak key in a TLS certificate occurred in 1,115 distinct certificates; the next most common keys appeared in 4 and 2 certificates.

4.2 Factorable RSA keys

A second way that entropy problems might manifest themselves during key generation is if an RSA modulus shares one of its prime factors p or q with another key. As explained in Section 2.1, finding such a pair immediately allows an adversary to efficiently factor both moduli and obtain their respective private keys. In order to find such keys, we computed the GCD of all pairs of distinct RSA moduli by applying the algorithm described in Section 3.3 to a combined dataset consisting of the keys in our TLS and SSH scans and the EFF SSL Observatory data. Running the algorithm on the combined datasets allowed us to factor more keys than performing the computation separately for each, since with a larger collection of inputs the algorithm has a larger factor base to use on each key.

The 11,170,883 distinct RSA moduli yielded 2,314 distinct prime divisors, which divided 16,717 distinct public keys. This allowed us to obtain private keys for 23,576 (0.40%) of the TLS certificates in our scan data, which were served on 64,081 (0.50%) of the TLS hosts, and 1,013 (0.02%) of the RSA SSH host keys, which were served on 2,459 (0.027%) of the RSA SSH hosts.

The vast majority of the vulnerable keys appeared to be system-generated certificates and SSH host keys used by routers, firewalls, remote administration cards, and other types of headless or embedded network devices. Only two of the factorable TLS certificates had been signed by a browser trusted authority and both have expired. Some devices generated factorable keys both for TLS and SSH, and a handful of devices shared common factors across SSH and TLS keys.

We classified these factorable keys in a similar manner to the repeated keys. We clustered the factorable certificates and host keys together by prime divisor. We found that, in all but a small number of cases, the TLS certificates and SSH host keys divisible by a common factor all belonged to a particular manufacturer, which we were able to identify in most cases using the techniques described in Section 3.2. We then grouped these clusters by manufacturer and compared them to other certificates and host keys from the same device model.

We identified devices from 41 manufacturers in this way, which constituted 99% of the hosts that generated RSA keys we could factor. The devices range from 100% (576 of 576 devices) vulnerable to 0.01% vulnerable (2 out of 10,932). As with repeated keys, we would not expect to see well-generated cofactorable keys; any device model observed generating factorable keys should be treated as potentially vulnerable.

The majority of the devices serving factorable keys were Juniper Networks Branch SRX devices.[3] We identified 46,993 of these devices in our dataset, and we factored the keys for 12,688 (27%) of them. Of these keys, 7,510 (59%) share a single common divisor. The distribution of factors among its moduli is shown in Figure 3a.

The most remarkable devices were IBM Remote Server Administration cards and BladeCenter Management Modules, which displayed a distribution of factors unlike any of the other vulnerable devices we found. There were only 9 distinct prime factors that had been used to generate the keys for 576 devices. Each device's key was the product of two different primes from this list. The 36 possible moduli that could be generated with this process were roughly uniformly distributed, as shown in Figure 3b. In addition, this was the only device we observed to generate RSA moduli where *both* prime factors were shared with other keys.

[3]Juniper identified these models in private communication.

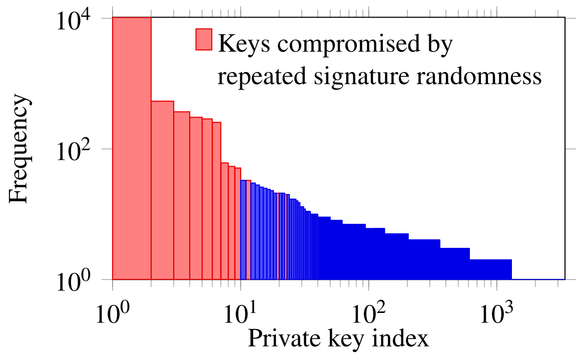


Figure 4: **Vulnerable DSA keys for one SSH device**— We identified 18,684 SSH DSA keys that appeared to have been generated by Gigaset DSL routers, of which 16,575 were repeated at least once. Shown in red in this log-log plot are 12,378 keys further compromised due to repeated DSA signature values. The more commonly repeated DSA keys were more likely to be compromised by repeated signatures, as they are more likely to have collisions between hosts in subsequent scans.

Several of the prime factors of the vulnerable keys displayed evidence of further entropy problems during key generation. Some shared an improbably large number of most significant bits in common—86 Sentry devices had common divisors sharing 160 out of 512 bits and two 2wire devices had a pair of common divisors with their 85 most significant bits in common. Alarmingly, we observed several prime factors of SSH host keys that were almost all zero bytes, with only the first two and last three bytes set, but we could not identify the device models. In addition to signaling that the primes themselves were generated with insufficient entropy, these constitute a separate, serious vulnerability, as it is possible to efficiently factor an RSA modulus if a large enough fraction of the bits of one of its factors is known [15] or shared with another modulus [41].

Approximately 12 of the SSH RSA keys were divisible by many small prime factors. Based on unlikely bit patterns in the keys, we hypothesize that these cases were due to memory corruption on the devices rather than broken primality testing.[4]

4.3 DSA signature weaknesses

The third class of entropy-related vulnerability that we searched for was repeated ephemeral keys in DSA signatures. As explained in Section 2.2, if a DSA key is used to sign two different messages using the same ephemeral

[4]The EFF SSL Observatory data contains several keys divisible by many small prime factors, but we determined that those were due to data corruption during scanning, as none of the cases was repeatable when the hosts were rescanned.

key, then the long-term private key is immediately computable from the public key and the signatures. The presence of this vulnerability indicates entropy problems at later phases of operation, after initial key generation. We searched for signatures from identical keys containing repeated *r* values. Then we used the method in Section 2.2 to compute the corresponding private keys.

Our combined SSH DSA scans collected 9,114,925 signatures (in most cases, two from each SSH host serving a DSA-based key) of which 4,365 (0.05%) contained the same *r* value as at least one other signature. 4,094 of these signatures (94%) used the same *r* value and the same public key. This allowed us to compute the 281 distinct private keys used to generate these signatures. These compromised keys were served by 105,728 (1.6%) of the SSH DSA hosts in our combined scans.

We clustered the vulnerable signatures by *r* values and manufacturer. 2,026 (46%) of the colliding *r* values appeared to have been generated by Gigaset SX762 consumer-grade DSL routers and revealed private keys for 17,349 (66%) of the 26,374 hosts we identified as this device model (see Figure 4). Another 934 signature collisions appeared to be from ADTran Total Access business-grade phone/network routers and revealed private keys for 62,807 (73%) out of 86,301 such hosts. Several vulnerable device models, including the IBM RSA II remote administration cards, also generated factorable RSA keys.

In addition to device randomness issues, we also discovered compromised keys due to flawed software implementations of DSA running on general-purpose servers. For example, hosts using the Java SSH Maverick library always used the same *r* value for every DSA signature.

We did not collect DSA handshake signatures during our TLS scan, but we did check the 6,107 DSA signatures that were applied to TLS certificates by signature authorities or self-signing. We found no repeated DSA public keys or signature ephemeral keys in this limited sample.

While we collected multiple signatures from some SSH hosts, 3,917 (89.7%) out of 4,365 of the collisions were from *different* hosts that had generated the same long-term key and also used the same ephemeral key during the key exchange protocol. This problem compounds the danger of the repeated key vulnerability: a single signature collision between any pair of hosts sharing the same key at any point during runtime reveals the private key for every host using that key. In our dataset we observed tens of thousands of hosts using the same public key. While a single host may never repeat an ephemeral key, two hosts sharing a private key can put each others' keys at risk.

We note that any estimation of vulnerability based on our data is an extreme lower bound, as we gathered at most two signatures from each host in our scans. It is likely that many more private keys would be revealed if we collected additional signatures.

5 Experimental Investigation

Based on the results the previous section, we conjectured that the problems we observed were an implementational phenomenon. To more deeply understand the causes, we augmented our data analysis with experimental investigation of specific implementations. While there are many independently vulnerable implementations, we chose to examine three open-source cryptographic software components that appeared frequently in the vulnerable populations.

5.1 Weak entropy and the Linux RNG

We conjectured that the cause for many of the entropy problems we observed began with insufficient randomness provided by the operating system. This led us to take an in-depth look at the Linux random number generator (RNG). We note that not every vulnerable key was generated on Linux; we also observed vulnerable keys on FreeBSD and Windows systems, and similar vulnerabilities to those we describe here have been reported with BSD's `arc4random` [50].

The Linux RNG maintains three entropy pools, each with an associated counter that estimates how much fresh entropy it has available. The RNG provides a function to mix data into the pool (using a twisted generalized feedback shift register) and to extract entropy from a pool (by hashing the pool contents, mixing the hash back into the pool, then hashing the hash again). Fresh entropy from unpredictable kernel sources is periodically mixed into the *Input pool*. When processes read from `/dev/random`, the kernel extracts the requested amount of entropy from the Input pool and mixes it into the *Blocking pool*, then extracts bytes from the Blocking pool to return to the process. If the Input pool does not contain enough entropy to satisfy the request, the read blocks until additional entropy becomes available. Similarly, when processes read from `/dev/urandom`, the kernel attempts to transfer the requested amount of entropy from the Input pool to the *Nonblocking pool*. In this case, the read will be satisfied immediately using the contents of the Nonblocking pool, even if the Input pool entropy is depleted.

Entropy sources We experimented with the Linux 2.6.35 kernel to exhaustively determine the sources of entropy used by the RNG. To do this, we traced through the kernel source code and systematically disabled entropy sources until the RNG output was repeatable. All of the entropy sources we found are greatly weakened under certain operating conditions.

When the kernel starts, the buffers that contain the pools are deliberately left uninitialized and may contain data left over from the previous boot; however, their contents will be predictable if the system has been pow-

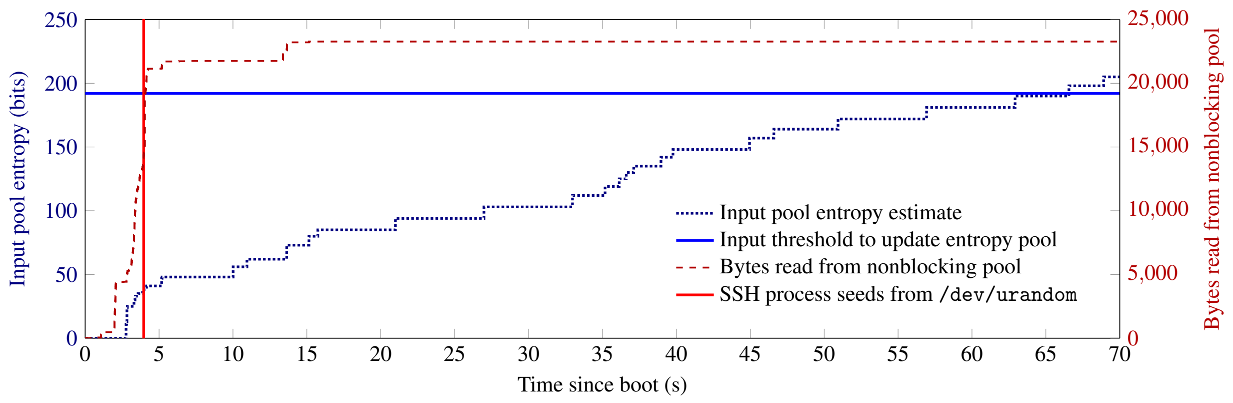


Figure 5: **Linux urandom boot-time entropy hole** — We instrumented an Ubuntu Server 10.04 system to record its estimate of the entropy contained in the Input entropy pool during a typical boot. Linux does not mix Input pool entropy into the Nonblocking pool that supplies `/dev/urandom` until the Input pool exceeds a threshold of 192 bits (blue horizontal line), which occurs here at 66 seconds post-boot. For comparison, we show the cumulative number of bytes generated from the Nonblocking entropy pool; the vertical red line marks the time when OpenSSH seeds its internal PRNG by reading from `urandom`, *well before* this facility is ready for secure use.

ered off long enough for memory to return to its ground state [27] or in systems with ECC RAM, which is zeroized on startup. The startup clock time, in nanosecond resolution, is also mixed in, but this value provides little entropy in systems that do not have a working real-time clock or on the first boot in systems where the real-time clock value is set at zero during manufacturing. It may also be predictable based on system uptime, which is visible to remote attackers via TCP timestamps.

Entropy from the timing of human input events and disk access is also mixed into the Input pool whenever these events occur. However, human input entropy is not available on unattended systems. Disk timings are only available relatively late in the kernel boot process, after disks have been initialized, and they are not available in embedded devices that lack traditional disk storage.

In addition, typical Linux distributions save entropy across boots by copying bytes from `urandom` to a local file on shutdown and reading them into the Blocking and Nonblocking pools at startup. However, this file is not present on the first system boot, and it will not be available until after the filesystem has been mounted.

The final and most interesting entropy source was one that we have not seen documented elsewhere. Whenever the kernel extracts entropy from a pool, it hashes the pool contents and then mixes part of the result back into the pool. The developers chose not to put a lock around this entire procedure, and, as a result, if two threads extract entropy concurrently, the pool contents may change anywhere in the middle of the hash computation. With eight physical threads, we observed on average approximately 100 instances of re-entrant entropy extraction from the Nonblocking pool during startup, mainly from entropy consumption within the kernel. There was a high degree of variation to the exact timing of these calls, resulting in the introduction of significant (but uncredited) entropy to the pool. Reentrant calls dropped to almost zero when we forced the kernel to use only one physical thread (with the `maxcpus=0` flag), and we did not observe any entropy being induced.

Surprisingly, modern Linux systems no longer collect entropy from IRQ timings. The Linux kernel maintainers deprecated the use of this source in 2009 by removing the `IRQF_SAMPLE_RANDOM` flag, apparently to prevent events controlled or observable by an attacker from contributing to the entropy pools. Although mailing list discussions suggest the intent to replace it with new collection mechanisms tailored to each interrupt-driven source [21], as of Linux 3.4.2 no such functions have been added to the kernel. The removal of IRQs as an entropy source has likely exacerbated RNG problems in headless and embedded devices, which often lack human input devices, disks, and multiple cores. If they do, the only source of entropy—if there are any at all—may be the time of boot.

Experiment To test whether Linux's `/dev/urandom` can produce repeatable output in conditions resembling the initial boot of a headless or embedded networked device, we modified the 2.6.35 kernel to add instrumentation to the RNG and disable certain entropy sources: (1) we zeroized the entropy pools during initialization to simulate a cold boot with constant memory state; (2) we used a fixed time value instead of the actual realtime clock value to simulate a machine without a working clock; (3) we limited the kernel to one physical thread to simulate a low-end CPU.

We experimented with this kernel on a Dell Optiplex 980 system using a fresh installation of Ubuntu server 10.04.4. The machine was configured with a Core i7 CPU, 4 GB RAM, a 32 GB SSD, and a USB keyboard. It was attached to a university office LAN and obtained an IP address using DHCP. With this configuration, we performed 1,000 unattended boots. Each time, we read 32 bytes from `urandom` at the point in the initialization process where the SSH server would normally start. Under these conditions, we found that the output of `/dev/urandom` was entirely predictable and repeatable.

The kernel maintains a reserve threshold for the Input pool, and no data is copied into the Nonblocking pool until the Input pool has been credited with at least that much entropy (192 bits, for our kernel). Figure 5 shows the cumulative amount of entropy credited to the Input pool during a typical bootup from our tests. (Note that none of the entropy sources we disabled would have resulted in more entropy being *credited* to the pool.) The credited entropy does not cross this reserve threshold until more than a minute after boot, well after the SSH server and other startup processes have launched. Although Ubuntu tries to restore entropy saved during the last shutdown, this happens slightly *after* the point when `sshd` first reads from `urandom`.

With no entropic inputs, `urandom` produces a deterministic output stream.We did observe variation due to nondeterministic behavior at boot that influences exactly how much data has been read from the stream at the point that the SSH server starts; for instance, other startup processes launched concurrently with the server may also be vying to read from the device. Figure 6 shows where in the predictable `urandom` stream the SSH server read. We saw 27 different outputs during the 1,000 boots; the most frequent occurred 23.7% of the time.

Boot-time entropy hole The entropy sources we disabled would likely be missing in some headless and embedded systems, particularly on first boot. This means that there is a window of vulnerability—a boot-time entropy hole—during which Linux's `urandom` may be entirely predictable, at least for single-core systems. If processes generate long-term cryptographic keys with entropy gathered during this window, those keys are likely to be vulnerable. The risk is particularly high for unattended systems that ship with preconfigured operating systems and generate SSH or TLS keys the first time the respective daemons start during the initial boot. Likewise, if processes maintain their own entropy pools which are seeded from `urandom` during this window and then maintained without the addition of further system entropy, any cryptographic randomness extracted from those pools is likely to be vulnerable.

On stock Ubuntu systems, these risks are somewhat mitigated: TLS keys must be generated manually, and

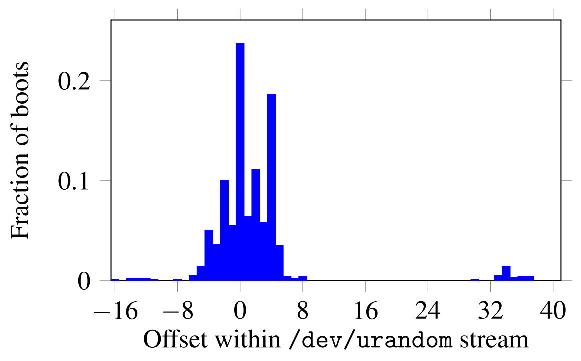


Figure 6: **Predictable output from urandom** — When we disabled entropy sources that might be unavailable on a headless or embedded device, the Linux RNG produced the same predictable stream on every boot. The only variation we observed over 1,000 boots was the *position* in the stream where `sshd` read from `urandom`.

OpenSSH host keys are generated during package installation, which is likely to be late in the install process, giving the system time to collect sufficient entropy. However, on the Fedora, Red Hat Enterprise Linux (RHEL), and CentOS Linux distributions, OpenSSH is installed by default, and host keys are generated on first boot. We experimented further with RHEL 5 and 6 to determine whether host keys on these systems might be compromised, and observed that sufficient entropy had been collected at the time of key generation (due to greater disk activity than with Ubuntu Server) by a slim margin. We believe that most server systems running these distributions are safe, particularly since they likely have multiple cores and gather additional entropy from physical concurrency. However, it is possible that other distributions and customized installations do not collect sufficient entropy on startup and generate weak keys on first boot.

5.2 Factorable RSA keys and OpenSSL

One interesting question raised by our vulnerability results is why factorable RSA keys occur at all. A naïve implementation of RSA key generation would simply seed a PRNG from the operating system's entropy pool and then use it to generate p and q. In this approach, we would expect to see duplicate keys if the OS provided the same seed, but factorable keys would be extremely unlikely. What we see instead is that some devices seem prone to generating keys with common factors. Another curious feature is that although some of the most common prime factors divided hundreds of different moduli, in nearly all of these cases the second prime factor did not divide any other keys.

One explanation for this pattern is that implementations updated their entropy pools in the middle of key

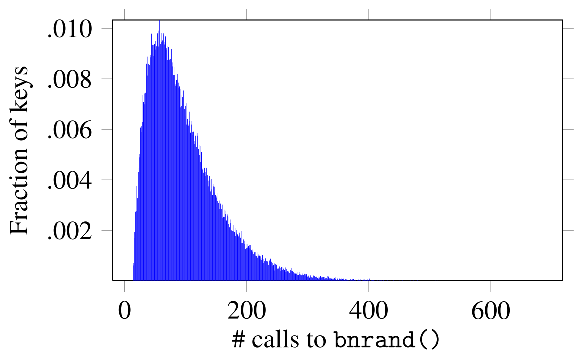


Figure 7: **Calls to bnrand() during key generation** — Every time OpenSSL extracts randomness from its internal entropy pool using `bnrand()`, it mixes the current time in seconds into the pool. This histogram shows the number of calls for 100,000 1024-bit RSA key generation operations. Both the average and the variance are relatively high due to the probabilistic procedure used to generate primes.

generation. In this case, the entropy pool states might be identical as distinct key generation processes generate the first prime p and diverge while generating the second prime q.

In order to explore this theory, we studied the source code of OpenSSL [16], one of the most widely used open-source cryptographic libraries. OpenSSL is not the only software library responsible for the problems we observed, but we chose to examine it because the source code is freely available and because of its popularity. SSH version strings identified OpenSSH (which uses OpenSSL's key generation libraries) for 75% of SSH hosts and 93% of SSH hosts with factored RSA keys. For TLS, OpenSSL was identified in the `nsComments` field of a certificate for 0.6% of TLS hosts and 0.6% of TLS hosts with factored RSA keys. Mironov [42] observes that OpenSSL's prime generation code contains tests to generate "safe" primes (those for which $p-1$ has no small factors) and that this unusual property for prime factors can be used to identify implementations that use this code. We found that 3,861 of the 16,717 factored moduli (23%) satisfied this property, suggesting that 46 of the 54 device clusters of factorable keys we identified use OpenSSL or similar code for generating "safe" primes.

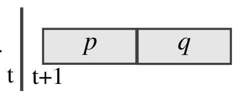
OpenSSL RSA key generation OpenSSL's built-in RSA key generator relies on an internal entropy pool maintained by OpenSSL. The entropy pool is seeded on first use with (on Linux) 32 bytes read from `/dev/urandom`, the process ID, user ID, and the current time in seconds. OpenSSL provides a function named `bnrand()` to generate cryptographically-sized integers from the entropy pool, which, on each call, mixes into the entropy pool the current time in seconds, the process ID, and the possibly uninitialized contents of a destination buffer. Note that, in identically configured systems, the only inputs to the entropy pool that should be expected to diverge are `urandom`, the time the pool is seeded, and the time of each `bnrand()` call.

The RSA key generation algorithm generates the primes p and q using a randomized algorithm. It selects a random integer using `bnrand()` and uses trial division to search for a candidate prime within a range. Once a candidate prime is found, it is passed through several rounds of the Miller-Rabin primality test, each of which also queries `bnrand()` to generate randomness. If the primality test fails, the above process is repeated with a new random integer until a likely prime is found.

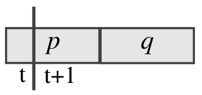
During this process, OpenSSL extracts entropy from the entropy pool dozens to hundreds of times, as shown in Figure 7. Since we observed many keys with one prime factor in common, we can conjecture that multiple systems are starting with `urandom` and the time in the same state and that the entropy pool states diverge due to the addition of time during intermediate steps of the key generation process.

We hypothesized that, even when two executions begin with identical inputs to the entropy pool and identical clocks, the key generation process is hypersensitive to small variations in where the *boundary between seconds* falls. Slight variations in execution speed might cause the wall clock tick to fall between different calls to `bnrand()`, resulting in different execution paths. This can result in several different behaviors, with three simple cases:

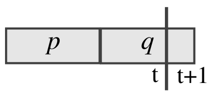
If the second never changes while computing p and q, every execution will generate identical keys.



If the clock ticks while generating p, both p and q diverge, yielding distinct keys with no shared factors.



If instead the clock advances to the next second during the generation of the second prime q, then two executions will generate identical primes p but can generate distinct primes q based on exactly when the second changes.



Experiment To test our hypothesis, we modified OpenSSL 1.0.0g to control all the entropy inputs used during key generation, generated a large number of RSA keys, and determined how many were identical or factorable. We set the process ID, user ID, and uninitialized input buffer to constant values and modified the `time()` function to simulate the process starting at a configurable time t_0. Our experimental machine was a Dell Optiplex 980 with a 2.80 GHz Core i7 processor running Ubuntu Server 10.04.4. For $t_0 \in \{0, 0.01, \ldots, 1.10\}$ we generated 100 1024-bit RSA keys using each of 40 different hard-

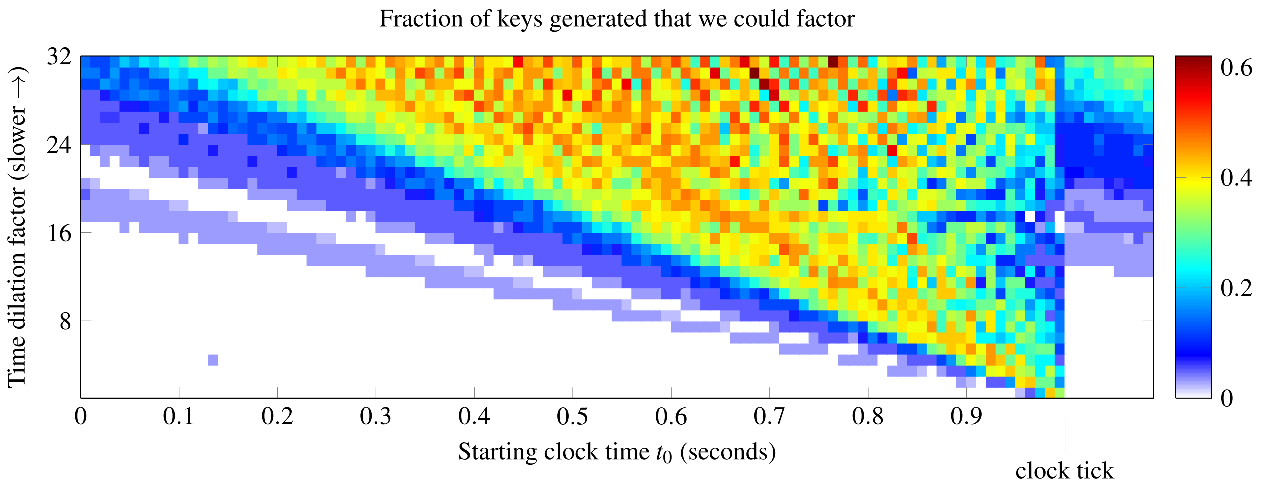


Figure 8: **OpenSSL generating factorable keys** — We hypothesized that OpenSSL can generate factorable keys under low-entropy conditions due to slight asynchronicity between the key generation process and the real-time clock, we generated 14 million RSA keys using controlled entropy sources for a range of starting clock times. Each square in the plot indicates the fraction of 100 generated keys that could we could factor. In many cases (white), keys are repeated but never share primes. After a sudden phase change, factorable keys occur during a range leading up to the second boundary, and that range increases as we simulate machines with slower execution speeds.

coded values in place of `/dev/urandom`. To simulate the effects of slower clock speeds, we dilated the clock time returned by `time()` and repeated the experiment using dilation multipliers of 1–32. In all, we generated 14 million keys. We checked for common factors within each batch of 100 keys.

The results we obtained, illustrated in Figure 8, are consistent with our hypothesis. At all time dilations, we find that factorable keys are generated during a range of start times leading up to the second boundary. No factorable keys are generated for low starting offsets, as both p and q are generated before the second changes. As the initial offset increases, there is a rapid phase change to generating factorable keys, as generation of q values begins to overlap the second boundary. Eventually, the fraction of factorable keys falls as the second boundary occurs during the generation of more p values, resulting in distinct moduli with no common factors.

The range of starting offsets when factorable keys occurred is narrow at fast clock speeds and spreads out as the time dilation factor increases and the simulated speed of the machine falls. For very slow simulated clocks, OpenSSL generates factorable keys for nearly the whole range of clock offsets. This may be reflective of the performance of many embedded devices.

While this theory can explain many of the device behaviors we observed, some devices produced distributions of factors that must have other causes. For example, the IBM RSA II remote administration devices (shown in Figure 3b) generated only nine distinct prime factors and the moduli appeared to be generated by selecting two of these prime factors uniformly at random. In this case, each modulus shared both of its prime factors with a large number of other devices. We saw evidence of this phenomenon only for a few hundred devices.

5.3 DSA signature weaknesses and Dropbear

The DSA signature vulnerabilities we observed indicate that entropy problems impact not only key generation but also the continued runtime behavior of server software during normal usage. This is somewhat surprising, since we might expect the operating system to collect sufficient entropy *eventually*, even in embedded devices. We investigated Dropbear, a popular light-weight SSH implementation. It maintains its own entropy pools seeded from the operating system at launch, on Linux with 32 bytes read from `urandom`. This suggests a possible explanation for the observed problems: the operating system had not collected enough entropy when the SSH server started, and, from then on, even though the system entropy pool may have had further entropy, the running SSH daemon did not.

To better understand why these programs produce vulnerable DSA signatures, we examined the source code for the current version of Dropbear, 0.55. The ephemeral key is generated as output from an internal entropy pool. Whenever Dropbear extracts data from its entropy pool, it increments a static counter and hashes the result into the pool state. No additional randomness is added until the counter (a 32-bit integer) overflows, at which point the entropy pool is reseeded with data from `urandom`, the current time, and the process ID. This implies that, if two Dropbear servers are initially seeded with the same value from `urandom`, they will provide identical signature randomness as long as their counters remain synchronized and do not overflow.

(We note that Dropbear contains a routine to generate k in a manner dependent on the message to be signed, which would ensure that distinct messages are always signed with distinct k values and protect against the vulnerability that we explore here. However, that code is disabled by default.)

We looked for evidence of synchronized sequences of ephemeral keys in the wild by making further SSH requests to a handful of the Dropbear hosts from our scan. We chose two hosts with the SSH version string `dropbear-0.39` that had used identical DSA public keys and r values during our original SSH scan. There were 356 IP addresses in our dataset that had served that public key, and we rescanned these addresses to find any that might currently be synchronized. We received 291 responses, and one pair used identical r values. We immediately collected 50 further signatures from these two hosts, and found that the signatures followed an identical sequence of r values. We could advance the sequence of one host by making several SSH requests, then cause the other host to catch up by making the same number of requests. When probed again an hour later, both hosts remained in sync. This suggests that the Dropbear code is causing vulnerabilities on real hosts in the manner we predicted.

Dropbear appeared in the version strings of 1,367,365 (13.4%) of the SSH hosts in our scans and 222 (5.09%) of the hosts with repeated DSA r values. Several other implementations, including hosts identifying OpenSSH and the Siemens Gigaset routers displayed similar behavior when rescanned. Because OpenSSL adds the current clock time to the entropy pool before extracting these random values, this suggests that some of these devices do not have a working clock at all.

6 Discussion

6.1 RSA vs. DSA in the face of low entropy

Any cryptosystem that relies on a secret key for security will be compromised if an adversary can determine that key. In practice, this might happen if an implementation leaks side-channel information about the key, or if the adversary can enumerate a reduced key space generated by low-entropy inputs. However, both RSA and DSA have specific catastrophic failure modes when implemented without sufficient entropy which can reveal the private key to an attacker who knows nothing about how the keys were generated. In a sense, the design of both cryptosystems permits the large-scale behavior of low-entropy implementations to act like a type of cryptographic side channel. From the point of view of robustness, these properties of the algorithms are less than ideal, but they result in vulnerability due to flawed implementations, not cryptographic flaws in either RSA or DSA.

We believe that the DSA signature vulnerabilities pose more cause for concern than the RSA factorization vulnerability. The RSA key factorization vulnerability that we investigated occurs only for certain patterns of key generation implementations in the presence of low entropy. In contrast, the DSA signature vulnerability can compromise any DSA private key—no matter how well generated—if there is ever insufficient entropy at the time the key is used for signing. It is not necessary to search for a collision, as we did; it suffices for an attacker to be able to guess the ephemeral private key k. The most analogous attacks against RSA of which we are aware show that some types of padding schemes can allow an attacker to discover the encrypted plaintext or forge signatures [11]. We are unaware of any attacks that use compromised RSA signatures to recover the private key.

We note that our findings show a larger fraction of SSH hosts are compromised by the DSA vulnerability than by factorable RSA keys, even though our scanning techniques have likely only revealed a small fraction of the hosts prone to repeating DSA signature randomness. In contrast, the factoring algorithm we used has found all of the repeated RSA primes in our sample of keys.

There are specific countermeasures that implementations can use to protect against these attacks. If both prime factors of an RSA modulus are generated from a PRNG without adding additional randomness during key generation, then low entropy would result in repeated but not factorable keys. These are more readily observable, but may be trickier to exploit, because they do not immediately reveal the private key to a remote attacker. To prevent DSA randomness collisions, the randomness for each signature can be generated as a function of the message and the pseudorandom input. (It is very important to use a cryptographically secure PRNG for this process [5].) Of course, the most important countermeasure is for implementations to use sufficient entropy during cryptographic operations that require randomness, but defense-in-depth remains the prudent course.

6.2 /dev/(u)random as a usability failure

Linux and some other UNIX-like operating systems provide two interfaces for generating randomness: `/dev/random`, which blocks until the system estimates that its entropy pool contains enough fresh entropy to satisfy the request, and `/dev/urandom`, which immediately returns the requested number of bytes even if no entropy has been added to the pool. Neither adequately suits the needs of application developers.

The Linux documentation states that "[a]s a general rule, urandom should be used for everything except long-lived GPG/SSL/SSH keys" [1]. However, all the open-source implementations we examined used urandom to generate keys by default. Based on a survey of developer mailing lists and forums, it appears that this choice is motivated by two factors: random's extremely conservative behavior and the mistaken perception that urandom's output is secure enough.

As others have noted, Linux is very conservative at crediting randomness added to the entropy pool [26], and random further insists on using *freshly collected* randomness that has not already been mixed into the output PRNG. The blocking behavior means that applications that read from random can hang unpredictably, and, in a headless device without human input or disk entropy, there may never be enough input for a read to complete. While blocking is intended to be an indicator that the system is running low on entropy, random often blocks even though the system has collected more than enough entropy to produce cryptographically strong PRNG output—in a sense, random is often "crying wolf" when it blocks.

This behavior appears to have contributed to a widespread belief among developers that random's blocking behavior is merely an annoyance to be worked around. A comment in the Dropbear source code reflects this view:

```
/* We'll use /dev/urandom by default,
since /dev/random is too much hassle.  If
system developers aren't keeping seeds
between boots nor getting any entropy from
somewhere it's their own fault.  */
#define DROPBEAR_RANDOM_DEV "/dev/urandom"
```

Our experiments suggest that many of the vulnerabilities we observed may be due to the output of urandom being used to seed entropy pools before any entropic inputs have been mixed in. Unfortunately, the existing interface to urandom gives the operating system no means of alerting applications to this dangerous condition. Our recommendation is that Linux should add a secure RNG that blocks until it has collected adequate seed entropy and thereafter behaves like urandom.

6.3 Are we seeing only the tip of the iceberg?

Nearly all of the vulnerable hosts that we were able to identify were headless or embedded devices. This raises the question of whether the problems we found appear *only* in these types of devices, or if instead we are merely seeing the tip of a much larger iceberg.

Based on the experiments described in Section 5.1, we conjecture that there may exist further classes of vulnerable keys that were not visible to our methods, but could be compromised with targeted attacks. The first class is composed of embedded or headless devices with an accurate real-time clock. In these cases, keys generated during the boot-time entropy hole may appear distinct, but depend only on a configuration-specific state and the boot time. These keys would not appear vulnerable in our scanning, but an attacker may be able to enumerate some or all of such a reduced key space for targeted implementations.

A more speculative class of potential vulnerability consists of traditional PC systems that automatically generate cryptographic keys on first boot. We observed in Section 5.1 that an experimental machine running RHEL 5 and 6 did collect sufficient entropy in time for SSH key generation, but that the margin of safety was slim. It is conceivable that some lower-resource systems may be vulnerable.

Finally, our study was only able to detect vulnerable DSA ephemeral keys under specific circumstances where a large number of systems shared the same long-term key and were choosing ephemeral keys from the same small set. There may be a larger set of hosts using ephemeral keys that do not repeat across different systems but are nonetheless vulnerable to a targeted attack.

We found no evidence suggesting that RSA keys from standard implementations that were generated interactively or subsequent to initial boot are vulnerable.

6.4 Directions for future work

In this work, we examined keys from two cryptographic algorithms on two protocols visible via Internet-wide scans of two ports. A natural direction for future work is to broaden the scope of all of these choices. Entropy problems can also affect the choice of Diffie-Hellman key parameters and keying material for symmetric ciphers. In addition, there are many more subtle attacks against RSA, DSA, and ECDSA that we did not search for. We focused on keys, but one might also try to search for evidence of repeated randomness in initialization vectors in ciphertext or salts in cryptographic hashes.

We also focused solely on services visible to our scans of the public Internet. Similar vulnerabilities might be found by applying this methodology to keys or other cryptographic data obtained from other resource-constrained devices that perform cryptographic operations, such as smart cards or mobile phones.

The observation that urandom can produce predictable output on some types of systems at boot may lead to attacks on other services that automatically begin at boot and depend on good randomness from the kernel. It warrants investigation to determine whether this behavior may undermine other security mechanisms such as address space layout randomization or TCP initial sequence numbers.

7 Defenses and Lessons

The vulnerabilities we have identified are a reminder that secure random number generation continues to be a challenging problem. There is a tendency for developers at each layer of the software stack to silently shift responsibility to other layers; a far better practice would be a defense-in-depth approach where developers at every layer apply careful security design and testing and make assumptions clear. We suggest defensive strategies and lessons for several important groups of stakeholders.

For OS developers:

Provide the RNG interface applications need. Typical security applications require a source of randomness that is guaranteed to be of high quality and has predictable performance; neither Linux's `/dev/random` nor `/dev/urandom` strikes this balance (see Section 6.2). The operating system should maintain a secure PRNG that refuses to return data until it has been seeded with a minimum amount of true randomness and is continually seeded with fresh entropy collected during operation.

Communicate entropy conditions to applications. The problem with `/dev/urandom` is that it can return data even before it has been seeded with any entropy. The OS should provide an interface to indicate how much entropy it has mixed into its PRNG, so that applications can gauge whether the state is sufficiently secure for their needs.

Test RNGs thoroughly on diverse platforms. Many of the entropy sources that Linux supports are not available on headless or embedded devices. These behaviors may not be apparent to OS developers unless they routinely test the internals of the entropy collection subsystem across the full spectrum on platforms the system supports.

For library developers:

Default to the most secure configuration. Both OpenSSL and Dropbear default to using `/dev/urandom` instead of `/dev/random`, and Dropbear defaults to using a less secure DSA signature randomness technique even though a more secure technique is available as an option. In general, cryptographic libraries should default to using the most secure mechanisms available. If the library provides fallback options, the documentation should make the trade-offs clear.

Use RSA and DSA defensively. Crypto libraries can take specific steps to prevent weak entropy from resulting in the immediate leak of private keys due to co-factorable RSA moduli and repeated DSA signature randomness (see Section 6.1).

Heed warnings from the OS, and pass them on. If the OS provides a mechanism to signal that the entropy it has collected is insufficient for cryptographic use (such as blocking when `/dev/random` is read), do not proceed with key generation or other security-critical operations. Provide an interface to communicate these states to applications, and document the condition as security critical.

For application developers:

Generate keys on first use, not on install or first boot. When keys are generated on initial boot, little or no entropy may be available from the operating system. If keys must be generated automatically, it may be better to defer generation until the keys are needed (such as during the first SSH connection), by which point the system will have had more of a chance to collect entropy.

Heed warnings from below. If the OS or cryptography library being used raises a signal that insufficient entropy is available (such as blocking), applications should detect this signal and refuse to perform security-critical operations until the system recovers from this potentially vulnerable state. Developers have been known to work around low-entropy states by ignoring or disabling such warnings, with extremely dangerous results [25].

Frequently add entropy from the operating system. When using libraries that maintain their own entropy pools, periodically add fresh randomness from the operating system in order to take advantage of additional entropy collected by the system during use. This is especially important for long-running daemons that are typically launched at the boot, when little or no entropy may be available.

For device manufacturers:

Avoid factory-default keys or certificates. While some defense is better than nothing, default keys and certificates provide only minimal protection. A better approach is to generate fresh keys on the device after sufficient randomness has been collected or to force users to upload their own keys.

Seed entropy at the factory. Devices could be initialized with truly random seeds at the factory. Sometimes it is already necessary to configure unique state on the assembly line (such as to set MAC addresses), and entropy could be added at the same time.

Ensure entropy sources are effective. Embedded or headless devices may not have access to sources of randomness assumed by the operating system, such as user-input devices or disk timing. Device makers should ensure that effective entropy sources are present, and that these are being harvested in advance of cryptographic operations.

Test cryptographic randomness on the completed device. Device makers should not simply assume that standard OS functions or cryptographic libraries are functioning correctly. Rather, they should test the complete device to check for weak keys and other entropy-related failures. Device makers can repeat some of the entropy tests that we describe in this paper on a smaller scale by generating many keys under simulated usage conditions and checking for repeated keys, factorable moduli, and repeated signature randomness.

Use hardware random number generators when possible. Given the problems we have identified with entropy collection in software, security-critical devices should use a hardware random number generator for cryptographic randomness whenever possible. Simple, low-cost circuits can provide sufficient entropy to seed a PRNG [48], and Intel has started introducing this capability in new CPUs [47].

For certificate authorities:

Check for repeated, weak, and factorable keys Certificate authorities have a uniquely broad view of keys contained in TLS certificates. We recommend that they repeat our work against their certificate databases and take steps to protect their customers by alerting them to potentially weak keys.

For end users:

Regenerate default or automatically generated keys. Cryptographic keys and certificates that were shipped with the device or automatically generated at first boot should be manually regenerated. Ideally, certificates and keys should be generated on another device (such as a desktop system) with access to adequate entropy.

Check for known weak keys. We have created a key-check service (see Section 8) that individuals can use to check their TLS certificates and SSH host keys against our database of keys we have identified as vulnerable. Additionally, we will list security advisories and software updates reported to us at https://factorable.net.

For security and crypto researchers:

Secure randomness remains unsolved in practice. The fact that all major operating systems now provide cryptographic RNGs might lead security experts to believe that any entropy problems that still occur are the fault of developers taking foolish shortcuts. Our findings suggest otherwise: entropy-related vulnerabilities can result from complex interaction between hardware, operating systems, applications, and cryptographic primitives. We have yet to develop the engineering practices and principles necessary to make predictably secure use of unpredictable randomness across the diverse variety of systems where it is required.

Primitives should fail gracefully under weak entropy. Cryptographic primitives are usually designed to be secure under ideal conditions, but practice will subject them to conditions that are less than ideal. We find that RSA and DSA, with surprising frequency, are used in practice under weak entropy scenarios where, due to the design of these cryptosystems, the private keys are totally compromised. More attention is needed to ensure that future primitives degrade gracefully under likely failure modes such as this.

8 Online Key-check Service

It is impractical for individual system administrators or device owners to repeat our Internet-wide survey work in order to detect whether their keys were generated with weak entropy. Therefore we have created an online service that allows users to check their SSH host keys or TLS certificates against our dataset. The service is available at https://factorable.net.

Our key-check service accepts X.509 certificates and base64-encoded host keys, and it can automatically retrieve keys from given HTTPS URLs or SSH server hostnames. It checks the provided keys against our database of known factorable keys, Debian weak keys, and snakeoil keys. While we cannot make any guarantees that a provided key is safe, we can immediately identify keys and certificates that we know to be weak.

Our service informs the user that their key or certificate is vulnerable, but it is designed not to reveal data that could further an attack. We report whether the key is known to be weak or factorable and the number of IP addresses presenting the same certificate or the same key. However, we do not display *which* other hosts use the same key or co-factorable keys. We rate-limit requests to prevent attackers from extracting the entire collection of weak keys, though we note that an attacker could repeat our scanning work and obtain the same information.

Ideally, we would also like to check whether a previously unseen key has a known factorization by calculating its GCD with our set of known keys. It takes approximately 190 seconds to test whether a new key has known factors given our current set of 11 million distinct moduli and 17 μs computation time per modulus. It is not feasible to provide results in real time for any significant traffic load, but we are considering a system that would allow users to submit keys for offline factorization and send the results by email.

9 Related Work

HTTPS surveys The HTTPS public-key infrastructure has been a focus of attention in recent years, and researchers have performed several large-scale scans to measure TLS usage and CA behavior. In contrast, our study addresses problems that are mostly separate from the CA ecosystem.

Ristic published an SSL survey in July 2010 [44] examining hosts serving the Alexa top 1 million domain names and 119 million other domain registrations. The study found 900,000 hosts serving HTTPS and 600,000 valid certificates. The same year, the Electronic Frontier Foundation (EFF) and iSEC Partners debuted the SSL Observatory project [20] and released the largest public repository of TLS certificates. They scanned approximately 87% of the IPv4 address space on port 443 and downloaded the resulting X.509 certificates over a three-month period. They released two datasets, the larger of which (from December 2010) recorded 4.0 million certificates from 7.7 million HTTPS hosts. The authors used their data to analyze the CA infrastructure and noted several vulnerabilities. We owe the inspiration for our work to their fascinating dataset, in which we first identified several of the entropy problems we describe; however, we ultimately performed our own scans to have more up-to-date and complete data. Our scan data contains approximately 67% more IP addresses and 45% more certificates than the latest publicly available SSL Observatory scan.

In 2011, Holz et al. [30] scanned the Alexa top 1 million domains and observed TLS sessions passing through the Munich Scientific Research Network (MWN). Their study recorded 960,000 certificates and was the largest academic study of TLS data at the time. They report many statistics gathered from their survey, mainly focusing on the state of the CA infrastructure. We note that they examined repeated keys and dismissed them as "curious, but not very frequent." Yilek et al. [51] performed daily scans of 50,000 TLS servers over several months to track replacement time for certificates affected by the Debian weak key bug. Our count of Debian certificates provides another data point on this subject.

Problems with random number generation Several significant vulnerabilities relating to weak random number generation have been found in widely used software. In 1996, the Netscape browser's SSL implementation was found to use fewer than a million possible seeds for its PRNG [22]. In May 2008, Bello discovered that the version of OpenSSL included in the Debian Linux distribution contained a bug that caused keys to be generated with only 15 bits of entropy [6]. The problem caused only 294,912 distinct keys to be generated per key size during a two year period before the error was found [51].In August 2010, the EFF SSL Observatory [20] reported that approximately 28,000 certificates, 500 of which were signed by browser-trusted certificate authorities, were still using these Debian weak keys.

Gutmann [25] draws lessons about secure software design from the example of developer responses to an OpenSSL update intended to ensure that the entropy pool was properly seeded before use. He observes that many developers responded by working around the safety checks in ways that supplied no randomness whatsoever. The root cause, according to Gutmann, was that the OpenSSL design left the difficult job of supplying sufficient entropy to library users. He concludes that PRNGs should handle entropy-gathering themselves.

Gutterman, Pinkas, and Reinmann analyzed the Linux random number generator in 2006 [26]. In contrast to our analysis, which focuses on empirical measurement of an instrumented Linux kernel, theirs was based mainly on a review of the LRNG design. They point out several weaknesses from a cryptographic perspective, some of which have since been remedied. In a brief experimental section, they observe that the only entropy source used by the OpenWRT Linux distribution was network interrupts.Dorrendorf, Gutterman, and Pinkas reverse-engineered the Windows random number generator [19] and discovered that an attacker with access to one state of the generator could enumerate all past and future states.

Weak entropy and cryptography In 2004, Bauer and Laurie [2] computed the pairwise GCDs of 18,000 RSA keys from the PGP web of trust and discovered a pair with a common factor of 9, demonstrating that the keys had been generated with broken (or omitted) primality testing.

The DSA signature weakness we investigate is well known and appears to be folklore. In 2010, the hacking group fail0verflow computed the ECDSA private key used for code signing on the Sony PS3 after observing that the signatures used repeated ephemeral keys [13]. Several more sophisticated attacks against DSA exist: Bellare, Goldwasser, and Miccancio [5] show that the private key is revealed if the ephemeral key is generated using a linear congruential generator, and Howgrave-Graham and Smart [31] give a method to compute the private key from a fraction of the bits of the ephemeral key.

Ristenpart and Yilek [43] developed "virtual machine reset" attacks in 2010 that induce repeated DSA ephemeral keys after a VM reset, and they implement "hedged" cryptography to protect against this type of randomness failure. Hedged public key encryption was introduced by Bellare et al. in 2009 and is designed to fail as gracefully as possible in the face of bad randomness [4].

As we were preparing this paper for submission, an independent group of researchers uploaded a preprint [37] reporting that they had computed the pairwise GCD of RSA moduli from the EFF SSL Observatory dataset and a database of PGP keys. Their work is concurrent and independent to our own; we were unaware of these authors' efforts before their work was made public. They declined to report the GCD computation method they used. We responded by publishing a blog post [29] describing our GCD computation approach and summarizing some of the key findings we detail in this paper.

The authors of the concurrent work report similar results to our own on the fraction of keys that were able to be factored, and thus the two results provide validation for each other. In their paper, however, the authors draw very different conclusions than we do. They do not analyze the source of these entropy failures, and they conclude that RSA is "significantly riskier" than DSA. In contrast, we performed original scans that targeted SSH as well as TLS, and we looked for DSA repeated signature weaknesses as well as cofactorable RSA keys. We find that SSH DSA private keys are compromised by weak entropy at a higher rate than RSA keys, and we conclude that the fundamental problem is an implementational issue rather than a cryptographic one.

Furthermore, the authors of the concurrent work state that they "cannot explain the relative frequencies and appearance" of the weak keys they observed and report no attempt to determine their source. In this work, we performed extensive investigation to trace the vulnerable keys back to specific devices and software implementations, and we have notified the responsible developers and manufacturers. We examined the source code of several of these implementations and performed experiments to understand why entropy problems are occurring. We find that the weak keys can be explained by specific design and implementation failures at various levels of the software stack, and we make detailed recommendations to developers and users that we hope will lessen the occurrence of these problems in the future.

10 Conclusion

In this work, we investigated the security of random number generation on a broad scale by performing and analyzing the most comprehensive Internet-wide scans of TLS certificates and SSH host keys to date. Using the global view provided by our data, we discovered that insecure RNGs are in widespread use, leading to a significant number of vulnerable RSA and DSA keys.

Our experiences suggest that the type of scanning and analysis we performed can be a useful tool for finding subtle flaws in cryptographic implementations, and we hope it will be applied more broadly in future work. Previous examples of random number generation flaws were found by painstakingly reverse engineering individual devices or implementations, or through luck when a collision was observed by a single user. Our scan data allowed us to essentially mine for vulnerabilities and detect problems in dozens of different devices and implementations in a single shot. Many of the collisions we found were too rare to ever have been observed by a single user but quickly became apparent with a near-global view of the universe of public keys. The results are a reminder to all that vulnerabilities can sometimes be hiding in plain sight.

Acknowledgments

The authors thank Dan Bernstein and Tanja Lange for discussion of batch factorization and OpenSSL, and Hovav Shacham for advice on many aspects of this work. We also thank Jake Appelbaum, Michael Bailey, Kevin Borders, Keith Brautigam, Ransom Briggs, Jesse Burns, Aleksander Durumeric, Prabal Dutta, Peter Eckersley, Andy Isaacson, James Kasten, Ben Laurie, Stephen Schultze, Ron Rivest, and David Robinson.

This material is based upon work supported by the National Science Foundation under Award No. DMS-1103803, the MURI program under AFOSR Grant No. FA9550-08-1-0352, and a National Science Foundation Graduate Research Fellowship.

References

[1] random(4) Linux manual page. http://www.kernel.org/doc/man-pages/online/pages/man4/random.4.html.

[2] BAUER, M., AND LAURIE, B. Factoring silly keys from the keyservers. In *The Shoestring Foundation Weblog* (July 2004). http://shoestringfoundation.org/cgi-bin/blosxom.cgi/2004/07/01#non-pgp-key.

[3] BEAZLEY, D. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proc. 4th USENIX Tcl/Tk Workshop* (1996), pp. 129–139.

[4] BELLARE, M., BRAKERSKI, Z., NAOR, M., RISTENPART, T., SEGEV, G., SHACHAM, H., AND YILEK, S. Hedged public-key encryption: How to protect against bad randomness. In *Proc. Asiacrypt 2009* (Dec. 2009), M. Matsui, Ed., pp. 232–249.

[5] BELLARE, M., GOLDWASSER, S., AND MICCIANCIO, D. "Pseudo-random" generators within cryptographic applications: the DSS case. In *Advances in Cryptology—CRYPTO '97* (Aug. 1997), B. S. Kaliski Jr., Ed., pp. 277–291.

[6] BELLO, L. DSA-1571-1 OpenSSL—Predictable random number generator, 2008. Debian Security Advisory. http://www.debian.org/security/2008/dsa-1571.

[7] BERNSTEIN, D. J. How to find the smooth parts of integers. http://cr.yp.to/papers.html#smoothparts.

[8] BERNSTEIN, D. J. Fast multiplication and its applications. *Algorithmic Number Theory* (May 2008), 325–384.

[9] BLUM, M., AND MICALI, S. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput. 13*, 4 (1984), 850–864.

[10] BONEH, D. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS 46*, 2 (1999), 203–213.

[11] BRIER, E., CLAVIER, C., CORON, J., AND NACCACHE, D. Cryptanalysis of RSA signatures with fixed-pattern padding. In *Advances in Cryptology—Crypto 2001*, pp. 433–439.

[12] BROWN, D. R. L. Standards for efficient cryptography 1: Elliptic curve cryptography, 2009. http://www.secg.org/download/aid-780/sec1-v2.pdf.

[13] BUSHING, MARCAN, SEGHER, AND SVEN. Console hacking 2010: PS3 epic fail. Talk at 27th Chaos Communication Congress (2010). http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf.

[14] CHOR, B., AND GOLDREICH, O. Unbiased bits from sources of weak randomness and probabilistic communication complexity. In *Proc. 26th IEEE Symposium on Foundations of Computer Science* (1985), pp. 429–442.

[15] COPPERSMITH, D. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology 10*, 4 (1997), 233–260.

[16] COX, M., ENGELSCHALL, R., HENSON, S., LAURIE, B., ET AL. The OpenSSL project. http://www.openssl.org.

[17] DAVIS, D., IHAKA, R., AND FENSTERMACHER, P. Cryptographic randomness from air turbulence in disk drives. In *Advances in Cryptology—CRYPTO '94* (1994), pp. 114–120.

[18] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol, Version 1.2. RFC 5246.

[19] DORRENDORF, L., GUTTERMAN, Z., AND PINKAS, B. Cryptanalysis of the Windows random number generator. In *Proc. 14th ACM Conference on Computer and Communications Security* (2007), CCS '07, pp. 476–485.

[20] ECKERSLEY, P., AND BURNS, J. An observatory for the SSLiverse. Talk at Defcon 18 (2010). https://www.eff.org/files/DefconSSLiverse.pdf.

[21] GETZ, R. IRQF_SAMPLE_RANDOM question ... Linux Kernel Mailing List post. https://lkml.org/lkml/2009/4/6/283.

[22] GOLDBERG, I., AND WAGNER, D. Randomness and the Netscape browser. *Dr. Dobb's Journal 21*, 1 (1996), 66–70.

[23] GRANLUND, T., ET AL. The GNU multiple precision arithmetic library. http://gmplib.org/.

[24] GUTMANN, P. Software generation of random numbers for cryptographic purposes. In *Proc. 7th USENIX Security Symposium* (1998), pp. 243–257.

[25] GUTMANN, P. Lessons learned in implementing and deploying crypto software. In *Proc. 11th USENIX Security Symposium* (2002), pp. 315–325.

[26] GUTTERMAN, Z., PINKAS, B., AND REINMAN, T. Analysis of the Linux random number generator. In *Proc. 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 371–385.

[27] HALDERMAN, J. A., SCHOEN, S., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J., FELDMAN, A., APPELBAUM, J., AND FELTEN, E. Lest we remember: Cold boot attacks on encryption keys. In *Proc. 17th USENIX Security Symposium* (July 2008), pp. 45–60.

[28] HEFFNER, C., ET AL. LittleBlackBox: Database of private SSL/SSH keys for embedded devices. http://code.google.com/p/littleblackbox/.

[29] HENINGER, N., ET AL. There's no need to panic over factorable keys–just mind your Ps and Qs. *Freedom to Tinker weblog* (2012). https://freedom-to-tinker.com/blog/nadiah/new-research-theres-no-need-panic-over-factorable-keys-just-mind-your-ps-and-qs.

[30] HOLZ, R., BRAUN, L., KAMMENHUBER, N., AND CARLE, G. The SSL landscape—A thorough analysis of the X. 509 PKI using active and passive measurements. In *Proc. 2011 ACM SIGCOMM Internet Measurement Conference* (2011), pp. 427–444.

[31] HOWGRAVE-GRAHAM, N., AND SMART, N. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography 23*, 3 (2001), 283–290.

[32] IANA. IPv4 address space registry. http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml.

[33] KINDER, K. Event-driven programming with Twisted and Python. *Linux Journal 2005*, 131 (2005), 6.

[34] KLEINJUNG, T., AOKI, K., FRANKE, J., LENSTRA, A., THOMÉ, E., BOS, J., GAUDRY, P., KRUPPA, A., MONTGOMERY, P., OSVIK, D., TE RIELE, H., TIMOFEEV, A., AND ZIMMERMANN, P. Factorization of a 768-bit RSA modulus. In *Advances in Cryptology—CRYPTO 2010* (2010), T. Rabin, Ed., pp. 333–350.

[35] LAWSON, N. DSA requirements for random k value, 2010. http://rdist.root.org/2010/11/19/dsa-requirements-for-random-k-value/.

[36] LENSTRA, A., LENSTRA, H., MANASSE, M., AND POLLARD, J. The number field sieve. In *The development of the number field sieve*, A. Lenstra and H. Lenstra, Eds., vol. 1554 of *Lecture Notes in Mathematics*. 1993, pp. 11–42.

[37] LENSTRA, A. K., HUGHES, J. P., AUGIER, M., BOS, J. W., KLEINJUNG, T., AND WACHTER, C. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. http://eprint.iacr.org/2012/064.pdf.

[38] LOCKE, G., AND GALLAGHER, P. FIPS PUB 186-3: Digital Signature Standard (DSS). Federal Information Processing Standards Publication (2009).

[39] LYON, G. F. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[40] MATHEWSON, N., AND PROVOS, N. libevent—An event notification library. http://libevent.org.

[41] MAY, A., AND RITZENHOFEN, M. Implicit factoring: On polynomial time factoring given only an implicit hint. In *Public Key Cryptography—PKC 2009*, pp. 1–14.

[42] MIRONOV, I. Factoring RSA moduli. Part II, 2012. http://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/.

[43] RISTENPART, T., AND YILEK, S. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Proc. ISOC Network and Distributed Security Symposium* (2010).

[44] RISTIC, I. Internet SSL survey 2010. Talk at BlackHat 2010. http://media.blackhat.com/bh-ad-10/Ristic/BlackHat-AD-2010-Ristic-Qualys-SSL-Survey-HTTP-Rating-Guide-slides.pdf.

[45] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21* (Feb. 1978), 120–126.

[46] SIONG, N., AND TOIVONEN, H. M2Crypto Python interface to OpenSSL. http://sandbox.rulemaker.net/ngps/m2/.

[47] TAYLOR, G., AND COX, G. Behind Intel's new random-number generator. *IEEE Spectrum* (Sept. 2011).

[48] TOKUNAGA, C., BLAAUW, D., AND MUDGE, T. True random number generator with a metastability-based quality control. *IEEE Journal of Solid-State Circuits 43*, 1 (Jan. 2008), 78–85.

[49] VINCENT, M. TI-83 Plus OS signing key cracked. In *ticalc.org weblog* (July 2009). http://www.ticalc.org/archives/news/articles/14/145/145154.html.

[50] WOOLLEY, R., MURRAY, M., DOUNIN, M., AND ERMILOV, R. FreeBSD security advisory FreeBSD-SA-08:11.arc4random, 2008. http://lists.freebsd.org/pipermail/freebsd-security-notifications/2008-November/000117.html.

[51] YILEK, S., RESCORLA, E., SHACHAM, H., ENRIGHT, B., AND SAVAGE, S. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proc. 2009 ACM SIGCOMM Internet Measurement Conference*, pp. 15–27.

[52] YLONEN, T. SSH—secure login connections over the internet. In *Proc. 6th USENIX Security Symposium* (1996), pp. 37–42.

[53] YLÖNEN, T., AND LONVICK, C. The secure shell (SSH) protocol architecture. http://merlot.tools.ietf.org/html/rfc4251.