Ensuring High-Quality Randomness in Cryptographic Key Generation

Henry Corrigan-Gibbs[*]
Stanford University
henrycg@stanford.edu

Wendy Mu
Stanford University
wmu@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Bryan Ford
Yale University
bryan.ford@yale.edu

ABSTRACT

The security of any cryptosystem relies on the secrecy of the system's secret keys. Yet, recent experimental work demonstrates that tens of thousands of devices on the Internet use RSA and DSA secrets drawn from a small pool of candidate values. As a result, an adversary can derive the device's secret keys without breaking the underlying cryptosystem. We introduce a new threat model, under which there is a *systemic* solution to such randomness flaws. In our model, when a device generates a cryptographic key, it incorporates some random values from an *entropy authority* into its cryptographic secrets and then *proves* to the authority, using zero-knowledge-proof techniques, that it performed this operation correctly. By presenting an entropy-authority-signed public-key certificate to a third party (like a certificate authority or SSH client), the device can demonstrate that its public key incorporates randomness from the authority and is therefore drawn from a large pool of candidate values. Where possible, our protocol protects against eavesdroppers, entropy authority misbehavior, and devices attempting to discredit the entropy authority. To demonstrate the practicality of our protocol, we have implemented and evaluated its performance on a commodity wireless home router. When running on a home router, our protocol incurs a 1.7× slowdown over conventional RSA key generation and it incurs a 3.6× slowdown over conventional EC-DSA key generation.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Applications*; E.3 [**Data Encryption**]: Public key cryptosystems

[*]Work conducted while author was a staff member at Yale University.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516680.

Keywords

entropy authority; cryptography; key generation; RSA; DSA; entropy; randomness

1. INTRODUCTION

A good source of randomness is crucial for a number of cryptographic operations. Public-key encryption schemes use randomness to achieve chosen-plaintext security, key-exchange algorithms use randomness to establish secret session keys, and commitment schemes use randomness to hide the committed value. The security of these schemes relies on the unpredictability of the random input values, so when the "random" inputs are not really random, dire security failures result [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 26, 29, 32, 41].

Although the dangers of weak randomness have been part of the computer security folklore for years [26], entropy failures are still commonplace. In 2008, a single mistaken patch caused the OpenSSL distribution in all Debian-based operating systems to use only the process ID (plus a few other easy-to-guess values) as the seed for its pseudo-random number generator. This bug caused affected machines to select a 1024-bit RSA modulus from a pool of fewer than one million values, rather than the near-2^{1000} possible values [41]. By replaying the key generation process using each of the one million possible PRNG seeds, an adversary could recover the secret key corresponding to one of these weak public keys in a matter or hours or days.

Recent surveys [29, 32] of SSH and TLS public keys on the Internet demonstrate that hardware devices with poorly seeded random number generators have led to a proliferation of weak cryptographic keys. During the drafting of this paper, NetBSD maintainers discovered a bug caused by a "misplaced parenthesis" that could have caused NetBSD machines to generate cryptographic keys incorporating as few as 32 or 64 bits of entropy, instead of the 100+ expected bits [36]. Even more recently, a PRNG initialization bug in the Android operating system could have caused applications using the system to generate weak cryptographic keys [31].

Randomness failures continue to haunt cryptographic software for a number of reasons: the randomness "stack" in a modern operating system [39] is large and complex, there is no simple way to test whether a random number generator is really generating random numbers, and (at least in the context of cryptographic keys) there has never been a *systemic solution* to the randomness problem. The response to entropy failures has traditionally been *ad hoc*: each device manufacturer or software vendor patches RNG-related bugs in its own implementation (once discovered), without deploying techniques to prevent similar failures in the future. The quantity and severity of randomness failures suggests that this "fix the implementation" approach is grossly insufficient.

We offer the first systemic solution to the entropy problem in cryptographic key generation for public-key cryptosystems. In our protocol, a device generating a cryptographic keypair fetches random values from an *entropy authority* and incorporates these values into its cryptographic secrets. The device can later *prove* to third parties (e.g., a certificate authority or an SSH client) that the device's secrets incorporate the authority's random values, thus guaranteeing that the device's cryptographic keys are selected from a large enough pool of candidate values. Unlike certificate authorities in today's Internet, our entropy authorities are *not* trusted third parties: if the device has a strong entropy source, a malicious entropy authority learns no useful information about the device's secret key. We present versions of our protocol for RSA and DSA key generation and we offer proofs of security for each.

A subtlety of our solution is the threat model: under a traditional "global passive adversary" model, the adversary can completely simulate the view of a device that has a very weak entropy source. Thus, under the global passive adversary model, a device with a weak entropy source has no hope of generating strong keys. We propose an alternate threat model, in which the adversary can observe all communication *except for* one initial communication session between the device and the entropy authority. Under this more limited adversary model, which is realistic in many deployment scenarios, we can take advantage of an entropy authority to ensure the randomness of cryptographic keys.

The key generation protocols we present are useful both for devices with strong and weak entropy sources. In particular, if the device has a strong entropy source (the device can repeatedly sample from the uniform distribution over a large set of values), running the protocol *never weakens* the device's cryptographic keys. In contrast, if the device has a weak or biased entropy source, running the protocol can *dramatically strengthen* the device's keys by ensuring that its keys incorporate sufficient randomness. The device need not know whether it has a strong or weak entropy source: the same protocol is used in both cases.

A recent survey of public keys [29] suggests that embedded devices are responsible for generating the majority of weak cryptographic keys on the Internet. To demonstrate that our protocols are practical even on this type of computationally limited network device, we have evaluated the protocols on a \$70 Linksys home router running the dd-wrt [22] operating system. Our RSA key generation protocol incurs less than a 2× slowdown on the Linksys router when generating a 2048-bit key, and our RSA and DSA protocols incur no more than 2 seconds of slowdown on a laptop and a workstation. The DSA version of our protocol is compatible with both elliptic-curve and finite-field groups. Our protocols generate standard RSA and DSA keys which are, for a given bit-length, as secure as their conventionally generated counterparts.

In prior work, Juels and Guajardo [30] present a protocol in which a possibly malicious device generates an RSA key in cooperation with a certificate authority. Their protocol prevents a device from generating an *ill-formed keypair* (e.g., an RSA modulus that is the product of more than two primes). We consider a different threat model. We ensure that a device samples its keys from a distribution with high min-entropy, but we do not prevent the device from generating malformed keys. Under this new threat model, we achieve roughly a 25× performance improvement over the protocol of Juels and Guajardo (as measured by the number of modular exponentiations that the device must compute). Section 7 compares the two protocols and discusses other related work.

After introducing our threat model in Section 2, we describe our key generation protocols in Section 3 and present security proofs in Section 4. Section 5 summarizes our evaluation results and Section 6 discusses issues related to integrating our protocols with existing systems.

1.1 Why Other Solutions Are Insufficient

Before describing our protocol in detail, we discuss a few other possible, but unsatisfactory, ways to prevent networked devices from using weak cryptographic keys.[1]

Possible Solution #1: Fix the implementation. One possible solution to the weak key problem is to simply make sure that cryptography libraries properly incorporate random values into the cryptographic secrets that they produce. Unfortunately, bugs and bad implementations are a fact of life in the world of software, and the subtleties of random number generation make randomness bugs particularly common. Implementations that seed their random number generators with public or guessable values (e.g., time, process ID, or MAC address) [8, 15, 16, 18, 18], implementations that use weak random number generators [10, 11, 12, 20, 21], and implementations without a good source of environmental entropy [29] are all vulnerable.

The complexity of generating cryptographically strong random numbers, the overwhelming number of randomness failures in deployed software, and the difficulty of detecting these failures during testing all indicate that "fix the implementation" is an insufficient solution to the weak key problem. Given that some implementations will be buggy, there should be a way to *assure* clients that their TLS and SSH servers are using strong keys, even if the client suspects that the servers do not have access to a good source of random values.

Possible Solution #2: Simple entropy server. A second possible solution would be to have devices fetch some random values from an "entropy sever" and incorporate these values (along with some random values that the device picks) into the device's cryptographic secrets. As long as the adversary cannot observe the device's communication with the server, the server would provide an effective source of environmental entropy.

One problem with this approach comes in attributing blame for failures. If a device using an entropy server produces

[1]By *weak keys* we mean keys sampled from a distribution with much less min-entropy than the user expects. For example, a 224-bit EC-DSA key sampled from a distribution with only 20 bits of min-entropy is weak.

weak keys, the device might *blame the entropy server* for providing it with weak random values. In turn, the entropy server could claim that it provided the device with strong random values but that the device failed to incorporate them into the device's cryptographic secrets. Without some additional protocol, a third party will not be able to definitively attribute the randomness failure to either the device or the entropy server.

Possible Solution #3: Key database. A third possible technique to prevent devices from using weak keys would be to deploy a "key database" that contains a copy of every public key on the Internet. A non-profit organization could run this database, much as the Electronic Frontier Foundation maintains the SSL Observatory [23], a static database of public keys on the Internet.

Whenever a device with a potentially weak entropy source generates a new keypair, the device would send its new public key to the key database. If the database already contains that key (or if the database contains an RSA modulus that shares a factor with the new key), the device would generate a fresh key and submit it to the database. The device would continue this generate-and-submit process until finding a unique key. At the end of the process, the device would be guaranteed to have a key that is unique, at least amongst the set of keys in the database.

Unfortunately, this proposed solution would *obscure* the entropy problem without fixing it. An attacker could replay the entire key generation process using the known initial state of a device with a weak entropy source to learn the secret keys of that device. By creating a centralized database of (possibly weak) keys, such a solution would make it easier for attackers to find and compromise weak keys.

2. SYSTEM OVERVIEW

Our proposed solution to the weak key problem, pictorially represented in Figure 1, takes place between a *device*, an *entropy authority*, a *certificate authority*[2] (optionally), and a *client*. We describe the roles of each of these participants before outlining our threat model and the security properties of the scheme.

2.1 Participants

Device. The *device* is the entity generating the RSA or DSA keypair that we want to ensure is sufficiently random, even if the device does not have access to a strong internal entropy source. The device might be an embedded device (e.g., a commodity wireless home router), or it might be a full-fledged server. The device could use the keypair it generates to secure HTTPS sessions and to authenticate itself in SSH sessions.

Entropy authority (EA). The *entropy authority* is the participant responsible for ensuring that a device's keypair is selected with enough randomness (is sampled independently from a distribution with high enough min-entropy). Just as a certificate authority verifies the identifying information (name, address, etc.) on a user's public key, the entropy authority verifies the *randomness* of a user's public key.

[2]IETF documents [5] use the term *certification authority* but we will follow common usage and use *certificate authority*.

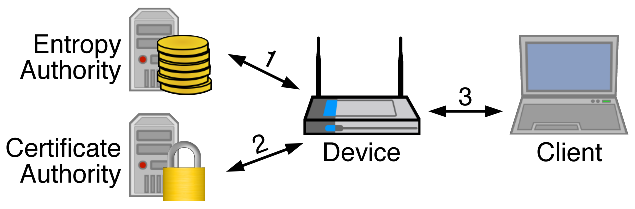

Figure 1: Overview of the protocol participants. (1) The device fetches random values from the entropy authority, proves to the authority that its key incorporates these values, and obtains a signature on the key from the EA. (2) The device submits its EA-signed public key to the certificate authority for signing. (3) The device presents an EA-signed key to a connecting client to prove that its keypair incorporates entropy from the authority.

As the device generates its cryptographic keypair, it fetches some random values from the entropy authority and then *proves* to the entropy authority that it has incorporated these values into its keypair. The entropy verifies this proof and then signs the device's public key. In practice, an entropy authority is just a public Web service with which the device interacts when the device first generates its a keypair. We assume that the entropy authority has a strong entropy source, but that the entropy authority might be malicious.

We imagine a future in which there are a large number of public entropy authorities on the Internet, run by corporate IT departments, certificate authorities, universities, and other large organizations. A device would select its entropy authority much as users select certificate authorities today: based on reputation and reliability. To defend against the failure (or maliciousness) of a single entropy authority, the device could interact with a number of entropy authorities to generate a single key, as we describe in Section 6.

Certificate authority (CA). The *certificate authority* plays the role of a conventional CA: the certificate authority confirms that the real-world identity of the device matches the identity listed in device's certificate, after which the CA signs the device's certificate. In our model, CAs will only sign certificates that have been signed first by an entropy authority. In this way, CAs are guaranteed to sign only public keys that are drawn from a distribution with high min-entropy. Since many certificates (particularly in embedded devices) are self-signed, the CA is an optional entity in our protocol.

Client. The *client* is anyone who connects to the device. In our model, the client can ensure that the device has a sufficiently random public key by verifying the entropy authority's signature on the key. Every client keeps a signature verification key for each entropy authority it trusts, just as today's Web browsers maintain a list of public keys for trusted root CAs.

2.2 Threat Model

Throughout the paper, we say that a device has a *strong entropy source* if it can repeatedly sample from the uniform distribution over some set (e.g., $\{0, 1\}$). We say that the device has a *weak entropy source* otherwise. A *strong key*, for our purposes, is a key independently sampled from a distribution over the set of possible keys that has at least

$k - \text{polylog}(k)$ bits of min-entropy, where k is the security parameter. In other words, a device generates strong keys if the probability that the device will generate a particular public key pk is at most $2^{-(k-\text{polylog}(k))}$ for all public keys pk. A *weak key* is any key that is not strong. We say that a participant is *honest* if it performs the protocol correctly and is *dishonest* otherwise.

The goal of our protocol is to have the device interact with the entropy authority in such a way that, after the interaction, the device holds a strong cryptographic key. This overall goal must be tempered a few realities. In particular, if a device has a no entropy source (or a very weak entropy source), then a *global* eavesdropper can always learn the device's secret key.

To see why this is so, consider that a device with no entropy source is just a deterministic process. Thus, the eavesdropper could always replay such a device's interaction with the entropy authority using the messages collected while eavesdropping. Thus, there is no hope for a completely deterministic device to generate keys that a global eavesdropper cannot guess.

To circumvent this fundamental problem, we consider instead a two-phase threat model:

1. *Set-up phase*: In the set-up phase, the device interacts with the entropy authority in a communication session that the adversary **cannot observe or modify**. In our key-generation protocols, this set-up communication session consists of two round-trip interactions between the device and the entropy authority.

2. *Long-term communication phase*: After the set-up stage ends, the adversary can observe and tamper with the traffic on all network links.

This threat model mimics SSH's implicit threat model: an SSH client gets one "free" interaction with the SSH server, in which the SSH server sends its public key to the client. As long as the adversary cannot tamper with this initial interaction, SSH protects against eavesdropping and man-in-the-middle attacks.

Under the adversary model outlined above, our key generation protocol provides the following security properties:

Protects device from a malicious EA. If the device has a strong entropy source, then the entropy authority learns no useful information about the device's secrets during a run of the protocol. We prove this property for the RSA protocol by demonstrating that the entropy authority can simulate its interaction with the device given only $O(\log k)$ bits of information about the RSA primes p and q. We prove this property for the DSA protocol by demonstrating that the entropy authority can perfectly simulate its interaction with the device given no extra information.

Protects device from CA and client. An honest device interacting with an honest entropy authority holds a strong key at the end of a protocol run, even if the device has a weak entropy source. When the device later interacts with a certificate authority (to obtain a public-key certificate) or with a client (to establish a TLS session), the device will send these parties a *strong* public key, even if the device has weak entropy source.

Protects EA from malicious device. If the entropy authority is honest, then the keys generated by this protocol will be strong, *even if* the device is dishonest. Intuitively, this property states that a faulty device cannot discard the random values that the entropy authority contributes to the key generation process.

A consequence of this security property is that a malicious device can never "discredit" an entropy authority by tricking the entropy authority into signing a key sampled from a low-entropy distribution. If a device does try to have the entropy authority sign a key sampled from a distribution with low min-entropy (a *weak key*), the authority will detect that the device misbehaved and will refuse to sign the key.

A nuance of this property is that the entropy authority *will accept public keys that are invalid*, as long as the keys are sampled independently from a distribution with high min-entropy. In essence, a faulty device in our protocol can create keys that are incorrect but random. For example, the device could pick an composite number as one of its RSA "primes," or it could use any number of other methods to "shoot itself in the foot" during the key generation process. Since the device can *always* compromise its own keypair (e.g., by publishing its secret key), we do not attempt to protect a completely malicious device from itself. Instead, we simply guarantee that any key that the entropy authority accepts will be drawn independently from a distribution with high min-entropy.

2.3 Non-threats

Our protocol addresses the threat posed by devices that use weak entropy sources to generate their cryptographic keys. We explicitly *do not* address these other broad vulnerability classes:

- **Adversarial devices.** If the device is completely adversarial, then the device can easily compromise its own security (e.g., by publishing its own secret key). Ensuring that such an adversarial device has high-entropy cryptographic keys is not useful, since *no* connection to such an adversarial device is secure.
- **Faulty cryptography library (or OS).** Our protocol does not attempt to protect against cryptographic software that is arbitrarily incorrect. Incorrect software can introduce any number of odd vulnerabilities (e.g., a timing channel that leaks the secret key), which we place out of scope.
- **Denial of service.** We do not address denial-of-service attacks by the entropy authority or certificate authority. In a real-world deployment, we expect that a device facing a denial-of-service attack by a CA or entropy authority could simply switch to using a new CA or EA.

3. PROTOCOL

This section describes a number of standard cryptographic primitives we require and then outlines our RSA and DSA key generation protocols.

3.1 Preliminaries

Our key generation protocols use the following cryptographic primitives.

Additively homomorphic commitments. We require an additively homomorphic and perfectly hiding commitment scheme. Given a commitment to x and a commitment to x', anyone should be able to construct a commitment to $x+x' \pmod{Q}$ without knowing the values x or x'. Our implementation uses Pedersen commitments [37]. Given public generators g, h of a group G with prime order Q, and a random value $r \in \mathbb{Z}_Q$, a Pedersen commitment to the value x is $\mathsf{Commit}(x; r) = g^x h^r$.[3] To ensure that the commitments are binding, participants must select the generators g and h in such a way that *no one* knows the discrete logarithm $\log_g h$.

The commitment scheme is additively homomorphic because the product of two commitments reveals a commitment to $x + x' \pmod{Q}$ with randomness $r + r' \pmod{Q}$:

$$\mathsf{Commit}(x + x'; r + r') = \mathsf{Commit}(x; r)\mathsf{Commit}(x'; r)$$

We abbreviate $\mathsf{Commit}(x; r)$ as $\mathsf{Commit}(x)$ when the randomness used in the commitment is not relevant to the exposition.

Of course, if the device has a weak entropy source the device will not be able to generate a strong random value r for use in the commitments. We use randomized commitments to hide a device's secrets *in case* the device does have a strong entropy source. Since a device does not necessarily know whether its randomness source is strong or weak, we must use the same constructions for devices with both strong and weak entropy sources.

Public-key signature scheme. We use a standard public-key signature scheme that is existentially unforgeable [27]. We denote the signing and verification algorithms by Sign and Verify.

Multiplication proof for committed values. We use a zero-knowledge proof-of-knowledge protocol that proves that the product of two committed values is equal to some third value. For example, given commitments C_x and C_y to values $x, y \in \mathbb{Z}_Q$, and a third product value $z \in \mathbb{Z}_Q$, the proof demonstrates that $z = xy \pmod{Q}$. We denote the prover and verifier algorithms by $\pi \leftarrow \mathsf{MulProve}(z, C_x, C_y)$ and $\mathsf{MulVer}(\pi, z, C_x, C_y)$.

We implement this proof using the method of Cramer and Damgård [6]. Written in Camenisch and Stadler's zero-knowledge proof notation [4], the multiplication proof proves the statement:

$$\mathsf{PoK}\{x, y, r_x, r_y, r_z : \\ C_x = g^x h^{r_x} \wedge C_y = g^y h^{r_y} \wedge g^z h^{r_z} = (C_x)^y h^{r_z}\}$$

Application of the Fiat-Shamir heuristic [24] converts this interactive zero-knowledge proof protocol into a non-interactive proof in the random-oracle model [2]. When implemented using a hash function that outputs length-l binary strings, the non-interactive multiplication proof is $l + 3\lceil \log_2 Q \rceil$ bits long.

Common Public Keys. We assume that all participants hold a signature verification public-key for the entropy and certificate authorities.

3.2 RSA Key Generation

The RSA key generation protocol takes place between the device and the entropy authority. At the end of a successful run of the protocol, the device holds an RSA public modulus n that is independently sampled from a distribution over $\mathbb{Z}$ that has high min-entropy and the device also holds the entropy authority's signature σ on this modulus.

In Section 4.1 we prove that the RSA protocol satisfies the security properties defined in Section 2.2. In Section 6, we describe how a device could use this protocol to generate a self-signed X.509 certificate and how to integrate this protocol with today's certificate authority infrastructure.

Parameters. Before the protocol begins, the device and entropy authority must agree on a set of common system parameters. These parameters include the security parameter k, which determines the bit-length of the RSA primes p and q. For a given value of k, the participants must also agree on a prime-order group G used for the Pedersen commitments and zero-knowledge proofs. The prime order Q of the group G must be somewhat larger than the largest RSA modulus n generated by the protocol, so the participants should let $Q \approx 2^{2k+100}$. In addition, participants must agree on two generators g and h of the group G, such that *no one* knows the discrete logarithm $\log_g h$. In an implementation of the protocol, participants could generate g and h using a shared public hash function. Finally, they also agree on a small number Δ (e.g. $\Delta = 2^{16}$) discussed in Section 3.2.1 below.

Since the parameters contain only public values, all devices and entropy authorities could share one set of parameters (per key size).

Protocol Description. Figure 2 presents our RSA key generation protocol. To generate an RSA key, the device first selects k-bit integers x and y and sends randomized commitments to these values to the entropy authority. The entropy authority then selects k-bit integers x' and y' at random and returns these values to the device.

After confirming that x' and y' are of the correct length, the device searches for offsets δ_x and δ_y such that the sums $p = x + x' + \delta_x$ and $q = y + y' + \delta_y$ are suitable RSA primes. That is, p and q must be distinct primes such that $\gcd(p-1, e) = 1$ and $\gcd(q-1, e) = 1$, where e is the RSA encryption exponent. The device then sets $n \leftarrow pq$, generates commitments to p and q, and produces a non-interactive zero-knowledge proof of knowledge π that the product of the committed values is equal to n. The device sends n, δ_x, δ_y, and the the proof π to the entropy authority.

The validity of the proof π and the fact that the δ values are less than Δ convince the entropy authority that the device's RSA primes p and q incorporate the authority's random values x' and y'. At this point, the authority signs the modulus n and returns it to the device.

3.2.1 Finding Primes p and q

To maintain the security of the protocol, it is important that the δ values chosen in Step 3 are relatively small—if the device could pick an arbitrarily large δ_x value, for example, the device could set $\delta_x \leftarrow -x'$, which would make $p = x + x' - x' = x$, thereby cancelling out the effect of the random value x' contributed by the entropy authority. To prevent the device from "throwing away" the entropy authority's entropy in this way, we require that the δ values be less than some maximum value Δ, which depends on the security parameter k.

[3]We denote the group order with capital "Q" to distinguish it from the RSA prime q in $n = pq$ that we use later on.

```
Device                                              Entropy Authority

Step 1
choose $x, y \xleftarrow{R} [2^k, 2^{k+1})$
$C_x \leftarrow \mathsf{Commit}(x)$
$C_y \leftarrow \mathsf{Commit}(y)$

                    send $C_x, C_y$ →
                                                    Step 2
                                    choose $x', y' \xleftarrow{R} [2^k, 2^{k+1})$

                    ← send $x', y'$
Step 3
abort if $x', y' \notin [2^k, 2^{k+1})$
choose $0 \leq \delta_x, \delta_y < \Delta$ s.t.
  $p \leftarrow x + x' + \delta_x$
  $q \leftarrow y + y' + \delta_y$
  are distinct primes,
  $\gcd(p-1, e) = 1$, and
  $\gcd(q-1, e) = 1$
abort if no such $\delta_x, \delta_y$ exist

let $n \leftarrow pq$
$C_p \leftarrow C_x g^{x'+\delta_x}$
$C_q \leftarrow C_y g^{y'+\delta_y}$
$\pi \leftarrow \mathsf{MulProve}(n, C_p, C_q)$

                    send $n, \delta_x, \delta_y, \pi$ →
                                                    Step 4
                                    $C_p \leftarrow C_x g^{x'+\delta_x}$
                                    $C_q \leftarrow C_y g^{y'+\delta_y}$

                                    abort if $\delta_x, \delta_y \notin [0, \Delta)$ or
                                    $n \notin [2^{2k+2}, 2^{2k+4})$ or
                                    $\mathsf{MulVer}(\pi, n, C_p, C_q) \neq 1$

                                    $\sigma \leftarrow \mathsf{Sign}_{\mathrm{EA}}(n)$

                    ← send $\sigma$
Step 5
abort if
  $\mathsf{Verify}_{\mathrm{EA}}(\sigma, n) \neq 1$

public key is $(n, e, \sigma)$
```

Figure 2: RSA Key Generation Protocol

Picking the size of Δ requires some care: if Δ is too small, then there may be no suitable prime p in the range $[x+x', x+x'+\Delta)$, and the device will have to run the protocol many times before it finds suitable primes p and q. The value Δ should be large enough that the protocol will succeed with overwhelming probability, but not so large that the device can pick $n = pq$ arbitrarily.

Following Juels and Guajardo [30], if the density of primes is d_{prime} and the density of these special primes (with $\gcd(p-1, q-1, e) = 1$ is d_{special}, we conjecture that $d_{\text{special}}/d_{\text{prime}} = (e-1)/e$, where e is the RSA encryption exponent (a small odd prime constant). Under this conjecture and the Hardy-Littlewood [28] conjecture, Juels and Guajardo demonstrate that the probability that there is *no suitable prime* in the interval $[x + x', x + x' + \Delta)$ is at most $\exp(-\lambda)$ when $\Delta = \lambda \ln(x + x')(\frac{e}{e-1})$ as $(x + x') \to \infty$. To make this conjecture concrete: if we take $(x + x') \approx 2^{1024}$, the RSA encryption exponent $e = 65537$, and require a failure probability of at most 2^{-80}, then we should set $\Delta \approx 2^{16}$. In the very unlikely case that the device fails to find primes p and q in the right range, the device aborts and re-runs the protocol from the beginning.

3.2.2 *Eliminating Information Leakage*

The values δ_x and δ_y sent to the entropy authority in Step 3 of the protocol leak some information about p and q to the entropy authority. In particular, the authority learns that the prime gap before p (resp. q) has a width of at least δ_x (resp. δ_y). We argue in Section 4.1 the entropy authority cannot use this leakage to help it factor the modulus n.

Even so, it is possible to modify the protocol to completely eliminate this information leakage at some performance cost. One way to modify the protocol is to require that $\delta_x = \delta_y = 0$ in Step 3 of the protocol. If the values $x + x'$ and $y + y'$ are not prime, the device aborts the protocol and restarts it from the beginning. Since the probability that a random k-bit number is a suitable prime is near $1/k$ for large k, the device will have to run the protocol approximately k^2 times before it succeeds.

To reduce the number of communication rounds required for this revised protocol, the device could run the k^2 protocol iterations in parallel. The device would send k^2-length vectors of commitments to random values $\vec{x}, \vec{y}$ in Step 1 of the protocol and the entropy authority would return two vectors $\vec{x}', \vec{y}'$ in Step 2 of the protocol, with each vector having length k^2. The device would then iterate over the vectors until it finds an i such that $p \leftarrow x_i + x'_i$ and $q \leftarrow y_i + y'_i$ are distinct primes and $\gcd(p-1, q-1, e) = 1$. If the device fails to find such primes, it would abort and repeat the process.

3.3 DSA Key Generation

The DSA key generation protocol, which we present in Figure 3, takes place between a device and the entropy authority.

Parameters. We assume that, before the start of the protocol, participants have agreed upon an order-q group G used in the DSA signing process. If the device uses the elliptic-curve variant of DSA (EC-DSA), then the group G will be an elliptic curve group selected, for example, from one of the NIST standard curves [25]. Participants must also agree upon two public generators, g and h, of the group G such that *no one* knows the discrete logarithm $\log_g h$.

While we expect most new devices to primarily use EC-DSA keys, even new devices may also need to generate finite-field DSA keys for interoperability with legacy devices. When using the finite-field variant of DSA, the device may have to generate the parameters of the finite-field DSA group (a prime modulus p, a group order q, and a generator g) in addition to its keypair. To do this, the device and entropy authority could agree on a *domain parameter seed* using a coin-flipping protocol [3] and then use this shared seed to generate DSA parameters using the verifiable generation method specified in the Digital Signature standard [25, Appendix A].

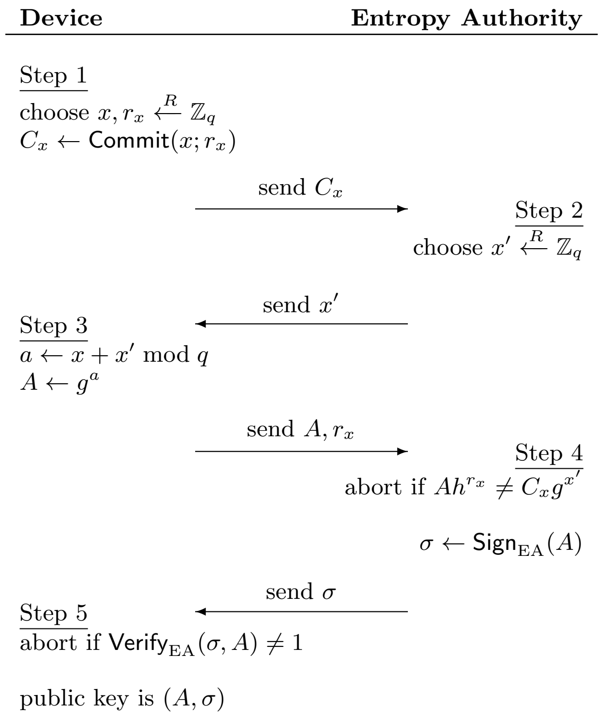
Device
Entropy Authority
Step 1
choose $x, r_x \xleftarrow{R} \mathbb{Z}_q$
$C_x \leftarrow \mathsf{Commit}(x; r_x)$
send C_x
Step 2
choose $x' \xleftarrow{R} \mathbb{Z}_q$
send x'
Step 3
$a \leftarrow x + x' \bmod q$
$A \leftarrow g^a$
send A, r_x
Step 4
abort if $Ah^{r_x} \neq C_x g^{x'}$
$\sigma \leftarrow \mathsf{Sign}_{\mathrm{EA}}(A)$
send σ
Step 5
abort if $\mathsf{Verify}_{\mathrm{EA}}(\sigma, A) \neq 1$
public key is (A, σ)

Figure 3: DSA Key Generation Protocol

Protocol Description. To begin the key generation process depicted in Figure 3, the device picks a random value $x \in \mathbb{Z}_q$ and generates a randomized commitment to x. In the event that the device has a strong entropy source, the use of a randomized commitment prevents the entropy authority from learning the device's secret x. The device sends this commitment to the entropy authority.

Upon receiving the device's commitment, the entropy authority chooses a random value $x' \in \mathbb{Z}_q$ and returns this value to the device. The device sets its private key $a \leftarrow x+x' \bmod q$ and sets its public key to $A \leftarrow g^a$. The device then sends its public key A along with the randomness r_x used in the commitment to x to the entropy authority.

The entropy authority confirms that $Ah^{r_x} = C_x g^{x'}$, which convinces the entropy authority that A is equal to $g^{x+x'}$. The entropy authority then signs the device's public key A and returns it to the device.

4. SECURITY ANALYSIS

This section presents proofs that the RSA and DSA key generation protocols satisfy the security properties described in Section 2.

4.1 RSA Protocol

4.1.1 Protects Device from a Malicious EA

We first show that when the device has a strong entropy source a malicious EA learns no useful information about the device's resulting RSA secret key.

First, let us define a standalone RSA modulus generation algorithm which does not interact with an EA. The key generator takes as input a security parameter k and lower bounds $p_{\min}$ and $q_{\min}$ on the RSA primes p and q.

```
$\mathsf{PrimeGen}(k, p_{\min})$:
    choose a random $x$ in $[2^k, 2^{k+1}]$
    find the smallest prime $p$ s.t. $p \geq p_{\min} + x$
    output $p$

$\mathsf{RSAKeyGen}(k, p_{\min}, q_{\min})$:
    $p \leftarrow \mathsf{PrimeGen}(k, p_{\min})$ , $q \leftarrow \mathsf{PrimeGen}(k, q_{\min})$
    output $n \leftarrow p \cdot q$
```

We say that a modulus generator outputs a *secure distribution of RSA moduli n* if the resulting family of RSA functions $x \to x^e \bmod n$ is a family of trapdoor one-way functions (where e is the RSA encryption exponent, a small prime constant). A secure modulus generator is sufficient for use in standard RSA encryption and RSA signature systems.

We use the following *RSA assumption* about algorithm $\mathsf{RSAKeyGen}$ above: algorithm $\mathsf{RSAKeyGen}(k, p_{\min}, q_{\min})$ outputs a secure distribution of RSA moduli for all $p_{\min}$ and $q_{\min}$ in the interval $[2^k, 2^{k+1})$.

The following theorem shows that even when interacting with a malicious EA, the RSA key generation protocol in Figure 2 outputs a secure distribution of RSA moduli. Furthermore, the protocol leaks at most $O(\log k)$ bits of information about the prime factors to the EA. This small leak does not harm security since if it were possible to invert the RSA function given the few leaked bits then it would also be possible to do it without, simply by trying all possible values for the leaked bits. Moreover, if desired this small leak can be eliminated at the cost of more computation, as explained in Section 3.2.2.

Theorem 4.1. *Suppose the device has a strong entropy source (i.e., the device can repeatedly sample independent uniform bits in $\{0,1\}$). Then for all EA, the protocol in Figure 2 generates a secure distribution of RSA moduli assuming the RSA assumption above. Furthermore, EA's view of the protocol can be simulated with at most $O(\log k)$ advice bits with high probability.*

Proof. Let $\mathcal{A}$ be a malicious EA that, given random commitments C_x, C_y, outputs $(x', y') \leftarrow \mathcal{A}(C_x, C_y)$. Then, since Pedersen commitments are information theoretically hiding, the protocol in Figure 2 outputs a modulus n sampled from the following distribution:

```
choose random $C_x, C_y \xleftarrow{R} \mathbb{Z}_Q$
$(p_{\min}, q_{\min}) \leftarrow \mathcal{A}(C_x, C_y)$
output $\mathsf{RSAKeyGen}(k, p_{\min}, q_{\min})$
```

Therefore, by the RSA assumption about algorithm $\mathsf{RSAKeyGen}$ the protocol generates secure distribution of RSA moduli.

Next, to argue that the protocol leaks at most $O(\log k)$ bits of information about the prime factors with high probability, we construct a simulator S that simulates the transcript of a successful run of the protocol with $\mathcal{A}$ given only n and an additional $O(\log k)$ bits of information. This will prove that given n, the protocol leaks only $O(\log k)$ additional bits. The protocol transcript consists of

$$(C_x, C_y, x', y', n, \delta_x, \delta_y, \pi, \sigma)$$

where $n = pq$ and $p = x + x' + \delta_x, \quad q = y + y' + \delta_y$ for some x, y. For a prime p let $\mathsf{pre}(p)$ be the prime immediately preceding p. The simulator S takes three arguments as input: the modulus $n = pq$ produced by a successful run of the protocol and the quantities

$$\Delta_p = \min(p - \mathsf{pre}(p),\ \Delta) \quad ; \quad \Delta_q = \min(q - \mathsf{pre}(q),\ \Delta)$$

The simulator works as follows:

```
$S(n, \Delta_p, \Delta_q)$:
    repeat:
        choose random $C_x, C_y \xleftarrow{R} \mathbb{Z}_Q$
        generate $(x', y') \leftarrow \mathcal{A}(C_x, C_y)$
    until $n \in \big[(x' + 2^k)(y' + 2^k),\ (x' + 2^{k+1})(y' + 2^{k+1})\ \big)$
    choose random $\delta_x$ in $[0, \Delta_p)$
    choose random $\delta_y$ in $[0, \Delta_q)$
    use the ZK simulator for Pedersen products to
        simulate a proof $\pi$ that $n = (x + x' + \delta_x)(y + y' + \delta_y)$
        where $x$ and $y$ are the values committed in $C_x, C_y$.
    run $\mathcal{A}$ giving it $n, \delta_x, \delta_y, \pi$ and obtain $\sigma$
    output the simulated transcript:
        $(C_x, C_y, x', y', n, \delta_x, \delta_y, \pi, \sigma)$
```

The simulator S properly simulates the Pedersen commitments C_x, C_y and the quantities x', y', given that the protocol generated the modulus n. Similarly, given that $n = pq$ was the output we know that the random variable $x + x'$ is uniformly distributed in the interval $(\mathsf{pre}(p), p]$ whenever $p - \mathrm{pre}(p) < \Delta$ and is uniform in $(p - \Delta, p]$ otherwise. Either way, the value of δ_x is uniform in $[0, \Delta_p)$. Hence S properly simulates δ_x and similarly δ_y. Finally, π is properly simulated using the ZK knowledge simulator for a proof of Pedersen products.

We explained in Section 3.2 that Δ_p and Δ_q are $O(k)$ in size with high probability, and therefore the protocol leaks at most $O(\log k)$ bits of information □

4.1.2 Protects Device from the CA and Client

Having established that the protocol protects a high-entropy device from the entropy authority, we demonstrate that an honest device interacting with an honest entropy authority holds a strong key at the end of a protocol run, even if the device has a weak entropy source.

THEOREM 4.2. *When interacting with an honest EA, the RSA protocol in Figure 2 generates a secure distribution of RSA moduli assuming the RSA assumption about algorithm* $\mathsf{RSAKeyGen}$.

PROOF. Let $\mathcal{A}$ be a device honestly following the protocol, but one that may have a weak entropy source. We let $\mathcal{A}()$ denote the x, y chosen by the device in Step 1. Given an honest EA, the protocol in Figure 2 outputs a modulus n sampled from the following distribution:

```
$(p_{\min}, q_{\min}) \leftarrow \mathcal{A}()$
output $\mathsf{RSAKeyGen}(k, p_{\min}, q_{\min})$
```

By the RSA assumption about algorithm $\mathsf{RSAKeyGen}$ the protocol generates secure RSA moduli. □

4.1.3 Protects EA from a Malicious Device

Suppose the device is *dishonest* and its goal is to discredit the entropy authority. The device may try to cause the EA to sign a modulus n in Step 4 of the protocol where n is sampled from a low entropy distribution. For example, the two prime factors of $n = pq$ may look non-random (e.g. their binary representation may end in many 1's) or n may have a non-trivial GCD with another public RSA modulus. The EA's signature would then serve as incriminating evidence that the "random" values x' and y' the EA contributed to the protocol in Step 2 were not sampled from the uniform distribution over $[2^k, 2^{k+1})$.

Note, however, that if the modulus n output by the device is an ill-formed RSA modulus—say n is not a product of two primes—then clearly the EA is not at fault since the device did not properly generate n. Therefore the EA need not worry about invalid moduli. It only cares about not signing low-entropy moduli.

The following theorem shows that an honest EA will never sign a low-entropy modulus.

THEOREM 4.3. *Consider an honest entropy authority interacting with a malicious polynomial-time device. Suppose that the RSA modulus n signed by an honest entropy authority in Step 4 is a product of two distinct primes $n = pq$ each in the range $[2^{k+1}, 2^{k+2})$. Then each of the primes is sampled from a distribution with at least $k - 2 - d\log(k)$ bits of min-entropy for some absolute constant d, even when conditioned on the other prime.*

PROOF SKETCH. We first show that n must be sampled from a distribution with sufficiently high min-entropy. In Step 4 of the protocol, the proof π convinces the EA that

$$n = (x + x' + \delta_x)(y + y' + \delta_y) \pmod{Q}$$

for some (unknown) x and y, where Q is the group order used for the Pedersen commitments. Recall that $Q > 2^{100}2^{2k}$. Since the device must commit to x before seeing x' we know that $x+x'$ is sampled independently from a distribution over $\mathbb{Z}_Q$ with min-entropy at least k (in the worst case, $x + x'$ is sampled uniformly from the integers in an interval of width 2^k). Since the device controls δ_x and $0 \le \delta_x < \Delta < c \cdot k$ for some absolute constant c, it follows that the min-entropy of $p_0 \leftarrow x + x' + \delta_x$ is at least $k - \log ck$. Similarly the min-entropy of $q_0 \leftarrow y + y' + \delta_y$ conditioned on p_0 is at least $k - \log ck$. Consequently, since n is a product of two primes, it can be shown for the distributions in question here that the min-entropy of $n = p_0 q_0 \bmod Q$ is at least $2k - d\log k$ for some constant d.

If n is the product of two primes $n = pq$ each in the range $[2^{k+1}, 2^{k+2})$ then each prime must be chosen from a distribution with min-entropy at least $k - 2 - d\log k$ (otherwise n cannot have min-entropy at least $2k - d\log k$). The theorem now follows. □

4.2 DSA Protocol

In this section, we prove that the DSA key generation protocol satisfies the security properties outlined in Section 2.

4.2.1 Protects Device from a Malicious EA

We first prove that a device with a strong entropy source leaks no information about its secret key to the entropy authority during a run of the protocol.

THEOREM 4.4. *If the device has a strong entropy source (i.e., the device can sample from the uniform distribution over a set), then the entropy authority can simulate its interaction with the device.*

PROOF. We construct a simulator S that, given a DSA public key $A = g^a$, outputs the transcript $(C_x, x', r_x, A, \sigma)$ of a protocol run between an honest device and a malicious entropy authority $\mathcal{A}$. The simulator S constructs the transcript as follows:

$S(A)$: choose random $r_x \xleftarrow{R} \mathbb{Z}_q$
set $C_x \leftarrow Ah^{r_x}$
generate $x' \leftarrow \mathcal{A}(C_x)$
run $\mathcal{A}$ on (A, r_x) to get a signature σ
output the simulated transcript: $(C_x, x', r_x, A, \sigma)$

The simulator's transcript is statistically indistinguishable from the transcript of a real protocol run between an honest device with a strong entropy source and the entropy authority $\mathcal{A}$. If the device has a strong entropy source, then the randomness r_x used in the commitment is sampled independently from the uniform distribution. The values C_x, x' and σ are constructed exactly as they would be in a real run of the protocol. $\square$

4.2.2 *Protects Device from the CA and Client*

Having established that a device with a strong entropy source leaks no secret information to the entropy authority, we now demonstrate that the key produced by the protocol is sampled from the uniform distribution over the set of possible keys.

THEOREM 4.5. *If an honest entropy authority does not abort the key generation protocol, and if* either *the device* or *the entropy authority has a strong entropy source, then the public keys produced by the protocol in Figure 3 will be sampled independently from the uniform distribution over* G.

PROOF. If the entropy authority does not abort the key generation protocol, then the equality $Ah^{r_x} = C_x g^{x'}$ is satisfied in the Step 4 of the protocol. Rearranging these terms to compute the public key A: $A = C_x g^{x'}/h^{r_x} = g^{x+x'}$. To demonstrate that the public key $g^{x+x'}$ is uniformly distributed over the set of possible public keys, it suffices to show that *either* x or x' is selected uniformly at random from $\mathbb{Z}_q$ and that x and x' are independent.

If the device has a strong entropy source, then the device's secret value x will be selected uniformly from $\mathbb{Z}_q$. To show that x' is independent of x, we rely on the "perfect hiding" property of the Pedersen commitment scheme. The commitment C_x that the entropy authority sees before selecting its value x' is simply a value $g^x h^{r_x}$ selected at random from G. If x is independent of x' and x is uniformly distributed over $\mathbb{Z}_q$ then $g^{x+x'}$ is uniformly distributed over G.

If the entropy authority has a strong entropy source, then the entropy authority's value x' will be selected uniformly from $\mathbb{Z}_q$. Since the device must commit to x *before* it sees x', the only way that the device's value x can depend on x' is if the device is able to open its commitment C_x to a value x that depends on x'. The "computationally binding" property of the Pedersen commitment scheme prevents the device from opening C_x to a value that depends on x'. If the device could open the commitment to a value of its choice, the device would be able to compute discrete logarithms in G. If x is independent of x' and x' is sampled from the uniform distribution over $\mathbb{Z}_q$ then $g^{x+x'}$ is sampled from the uniform distribution over G. $\square$

4.2.3 *Protects EA from a Malicious Device*

Theorem 4.2.2 holds *even if* the device is malicious, so a key produced from an interaction between a malicious device and an honest entropy authority will be strong.

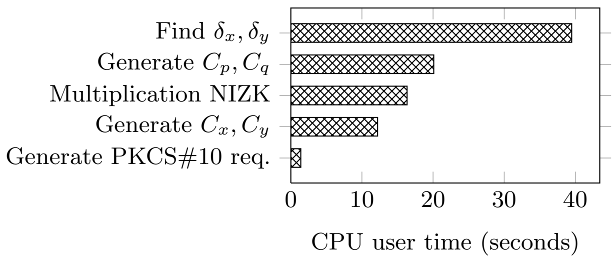

Figure 4: Operations taking longer than 0.05s during a run of the RSA protocol on the home router.

5. EVALUATION

To demonstrate the practicality of our RSA and DSA key generation protocols, we implemented the protocols in C using the OpenSSL cryptography library. We evaluated the performance of the protocols on three different devices: a Linux workstation with two 3.2 GHz Intel W3656 processors, a MacBook Pro laptop with a single 2.5 GHz dual-core processor, and a Linksys E2500-NP home router with a 300 MHz Broadcom BCM5357r2 processor. The entropy authority in all experiments was a modern Linux server and the DSA protocol experiments use the NIST P-224 elliptic curve as the elliptic curve DSA (EC-DSA) group [25].

Embedded devices, like the Linksys router we used in our evaluations, lack the keyboard, mouse, hard drive, and other peripherals used as entropy sources on full-fledged machines. As a result, these device are particularly susceptible to generating weak keys. By evaluating our key generation protocols on a $70 Linksys router, we demonstrate that the protocols are practical even on low-power, low-cost (and often low-entropy) embedded devices. For the purposes of evaluation, we installed the Linux-based dd-wrt [22] operating system on the Linksys router and ran our key generation protocol in a user-space Linux process.

Table 1 presents the wall-clock time required to generate a 2048-bit RSA key and a 224-bit EC-DSA key on each machine, averaged over eight trial runs. When running on the laptop and workstation, which have relatively fast CPUs, the bulk of the protocol overhead (roughly 90%) comes from the network latency in communicating with the entropy authority. On the CPU-limited home router, the protocol causes a near-2× slowdown, even without the network latency. Even so, running the EC-DSA protocol takes fewer than two seconds on all three of the devices.

The standard RSA keypair generation algorithm requires much more computation than the EC-DSA algorithm, so the cost of interacting with the entropy authority is amortized over a longer total computation in the RSA protocol. As a result, the slowdown factors on each of the three devices is smaller for the RSA protocol than for the DSA variant. The protocol incurs less than a 2× slowdown when running on the home router—generating a standard 2048-bit RSA keypair takes nearly 60 seconds and generating a keypair with the protocol takes just over 100 seconds. On the laptop and workstation, around 50% of the slowdown is due to network latency. On these faster devices, generating an RSA keypair using the protocol takes less than three seconds.

Figure 4 presents a graphical break-down of the CPU user time required to perform the most expensive operations in the RSA key generation protocol on the home router. Nearly half of the CPU time consumed during the protocol is spent

	EC-DSA (224-bit prime)				**RSA** (2048-bit)			
	No proto	Proto	Proto+Net	*Slowdown*	No proto	Proto	Proto+Net	*Slowdown*
Linksys Router	0.45	0.84	1.61	3.6×	59.16	96.93	101.57	1.7×
Laptop	0.03	0.08	0.72	28×	0.52	1.26	2.01	3.9×
Workstation	0.004	0.05	0.68	160×	0.16	0.65	1.41	8.7×

Table 1: Time (in seconds) to generate a keypair without our protocol, with a local EA, and with an EA via the Internet with ≈ 80 ms of round-trip latency. The Slowdown column indicates the slowdown factor of our protocol running over the Internet relative to the standard key generation algorithm.

in finding the δ_x and δ_y offset values to make the RSA factors p and q prime. Finding these offsets requires running the Miller-Rabin [38] primality test on a number of candidate primes. This expensive search for primes p and q is also required to generate an RSA modulus *without* our key generation protocol, so this search does not constitute protocol overhead.

The other expensive operations are computing the Pedersen commitments (each of which requires big-integer modular exponentiations) and generating the non-interactive zero-knowledge proof that n is the product of the values contained in the commitments C_p and C_q. The final expensive operation is generating the PKCS#10 certificate request, which the device signs with its newly generated RSA key.

Our implementation does not use fast multi-exponentiation algorithms [34] (e.g., for computing Pedersen commitments $g^a h^r$ quickly) or exploit parallelism to increase performance on multi-core machines. An aggressively optimized production-ready implementation could use these techniques to improve the performance of the protocol.

As shown in Figure 5, our protocol imposes a near-uniform 4× computation overhead (measured in CPU user time) on EC-DSA key generation. This slowdown arises because our EC-DSA protocol requires three elliptic curve point multiplications and a single signature verification, compared with the single elliptic curve point multiplication required in traditional EC-DSA key generation. At the smallest usable EC-DSA key size, 112 bits, the protocol set-up cost dominates the overall running time, so the protocol imposes a 5.6× overhead.

The computational overhead of generating RSA keys using our protocol decreases as the key size increases. The dominant *additional* cost of our RSA protocol is the cost of the modular exponentiations used in the commitment scheme and zero-knowledge proof generation. As k increases, the cost of finding the RSA primes grows faster than the additional cost of our protocol, so the computational overhead of our protocol tends to 1.

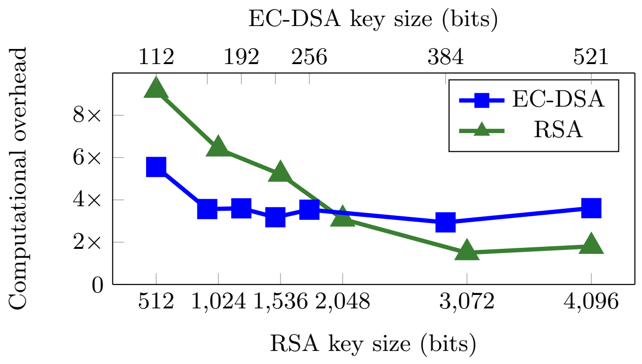

Figure 5: Computational overhead (in CPU user time) imposed when a laptop uses our key generation protocols to generate keypairs of various sizes.

6. IMPLEMENTATION CONCERNS

This section discusses a handful of practical implementation issues that a real-world deployment of our key generation protocols would have to address.

Integration with the CA infrastructure. Integrating our key generation protocols with the existing CA infrastructure would require only modest modifications to today's infrastructure. In a deployment of our key generation protocol, the device could interact with the entropy authority using an HTTP API. After the device obtains the entropy authority's signature on its public key, the device would embed the EA signature in an extension field in the PKCS#10 certificate signing request that the device sends to the certificate authority. Each certificate authority would maintain a list of public keys of approved entropy authorities (in the way that browsers and SSL libraries today maintain a list of root CA public keys). When a certificate authority receives a PKCS#10 request from a device, the CA would first check the validity of the EA's signature on the request. If the signature is valid and the CA is able to verify the identity of the requesting device, the CA would sign the certificate and return it to the device.

We expect that many commercial certificate authorities would be willing to serve as free public entropy authorities, since the computational cost of acting as an entropy authority is small (less than one CPU-second per protocol run). Organizations large enough to have their own IT departments might run their own internal entropy authorities as well.

Self-Signed Certificates and SSH. TLS servers often use self-signed certificates to provide link encryption without CA-certified identity. The analogue of a self-signed certificate in our setting is a certificate that is signed by the entropy authority but that is *not* signed by a certificate authority. This sort of certificate would convince a third party that the device's public key is sampled from a high-entropy distribution, without convincing a third party that the key corresponds to a particular real-world identity. As long as some EAs provide their services for free (which we expect), EA-signed certificates will be free, just as self-signed certificates are free today.

To generate such a certificate, the device would submit a PKCS#10 certificate signing request to the entropy authority at the end of Step 3 of the RSA protocol or Step 3 of the DSA protocol, along with other data it sends. The entropy authority would then sign the request and would return the EA-signed certificate to the device. TLS clients (e.g., Web browsers) would maintain a list of public keys of approved entropy authorities, just as today's client keep a list of approved root CAs. When a client connects to a device that uses an EA-signed certificate, the client would verify the EA's signature and would treat the certificate just as it treats self-signed certificates today.

SSH could similarly use EA-signed keys to use convince clients that a particular SSH host generated its public key using random values from an approved entropy authority. To accomplish this, the SSH server software would define a new public key algorithm type for EA-signed keys (e.g., `ssh-rsa-rand`). Keys of this type would contain the SSH host's normal public key, but they would also contain an EA's signature on the SSH host's public key (along with the fingerprint of the signing EA's key). SSH clients that support the `ssh-rsa-rand` key type would be able to verify the EA's signature on the host's key to confirm that that the host's key incorporates randomness from an approved entropy authority.

Other entropy issues. Our key generation protocol *only* ensures that a device's RSA or DSA keypair has sufficient randomness—it does not ensure randomness in other security-critical parts of the system (e.g., signing nonce generation, TLS session key selection, address space layout randomization). We focus on cryptographic key generation because attacks against weak public keys are especially easy to mount. Once a device publishes a weak public key, the device is likely to use the same public key for months or years. Thus, even if the device's entropy source strengthens over time (as the device gathers randomness from network interrupt timings, for example) the device's keys remains weak. Hedged public-key cryptography [1, 39], in conjunction with our key generation protocols, would help reduce the risk of bad randomness in signing and encryption, but solving all of these randomness problems is likely beyond the scope of any single system.

Distributing trust with many entropy authorities. As we note in Section 2, if the device has a weak entropy source then there is no way to protect the device against a eavesdropper that observes *all communication* between the device and the EA. Our threat model excludes the possibility of such an eavesdropper, but if the device is particularly concerned about eavesdroppers on its initial conversation with the EA, the device could run a modified version of the protocol with *many* entropy authorities instead of just one. With multiple EAs, an eavesdropper would have to observe the device's communication with *all* of the EAs to learn the device's secret key. We sketch the multi-authority DSA protocol here. A similar modification allows RSA key generation with multiple entropy authorities.

In the multi-authority DSA protocol, the device commits to its random value x and sends $C_x \leftarrow \mathsf{Commit}(x; r)$ to each of N entropy authorities. Each entropy authority responds with (x_i, r_i, σ_i), where x_i and r_i are random values in $\mathbb{Z}_q$ and $\sigma_i \leftarrow \mathsf{Sign}_{EA_i}(C_x, \mathsf{Commit}(x_i; r_i))$. The device's secret key is then $a = x + \Sigma_i x_i \bmod q$. The device can obtain a signature on its public key g^a from each entropy authority by presenting each authority with its public key, its commitment to x, the randomness r it used to commit to x, the set $\vec{r}$ of nonces used in the entropy authorities' commitments,

	Single exp.	Double exp.
Juels-Guajardo protocol [30]	319	35
This paper	8	4

Figure 6: Approximate number of k-bit modular exponentiations the device must compute to generate a k-bit RSA modulus.

the set $\vec{C}$ of commitments to each of the entropy authority's random values, and the set of entropy authority signatures $\vec{\sigma}$: $(g^a, C_x, r, \vec{r}, \vec{C}, \vec{\sigma})$. Each authority verifies each EA signature σ_i, confirms that $g^a h^{r+\Sigma_i r_i} = C_x(\Pi_i C_i)$, and signs the device's public key g^a.

Default keys. Roughly 5% of TLS hosts on the Internet in 2012 used *default keys*, which are pre-loaded into the device's firmware by the manufacturer [29]. Typically, any two such devices of the same model and firmware version will ship with the same public and secret key. To recover a default secret key, an attacker can download the firmware for the device from the manufacturer's Web site or look up the default key in a database designed for that purpose [33].

Our protocol does not protect against a manufacturer who installs the same keypair in many devices. If a manufacturer wants all of her devices to ship with a default keypair signed by an entropy authority, the manufacturer could run our key generation protocol *once* in the factory, and then install this single EA-signed keypair in every device shipped.

Installing the same keypair in many devices is tantamount to publishing the device's secret key, which is an "attack" which we cannot hope to prevent. As a heuristic defense against default keys, a client connecting to a device could require that the device use a certificate that was generated after the manufacture of the device (as indicated, for example, by an EA-signed timestamp on the certificate).

7. RELATED WORK

Hedged public-key cryptography [1, 39] addresses the problem of weak randomness during message *signing* or *encryption*, whereas our work addresses the problem of randomness during *key generation*. Cryptographic hedging provides no protection against randomness failures when generating cryptographic keys but deployed systems could use hedging in conjunction with our key-generation protocols to defend against weak randomness after generating their cryptographic keys.

Intel's Ivy Bridge processor implements a hardware instruction that exploits physical uncertainty in a dedicated circuit to gather random numbers [40]. A hardware random number generator provides a new and potentially rich source of entropy to cryptographic applications. Devices without hardware random number generators could use a variety of other techniques to gather possibly unpredictable values early in the system boot process [35]. Even so, having a rich entropy source does not mean that software developers will properly incorporate the entropy into cryptographic secrets. Our protocol ensures that keys will have high entropy, even if the cryptographic software ignores or misuses hardware-supplied randomness.

Juels and Guajardo [30] offer a protocol for RSA key generation that is superficially similar to the one we present here. The Juels-Guajardo protocol protects against *kleptography* [42], in which a device's cryptography library is adversarial, and *repudiation*, in which a signer intentionally generates a weak cryptographic signing key so that the signer can disown signed messages in the future. To prevent against these very strong adversaries, their protocol requires a number of additional zero-knowledge proofs that are unnecessary in our model. Using the number of modular exponentiations as a proxy for protocol execution time, the Juels-Guajardo protocol would likely take over 40 minutes to execute on the home router we used in our experiments, while our protocol takes fewer than two minutes (see Figure 6). In addition, Juels and Guajardo do *not* address the issue of a device whose source of randomness is so weak that it cannot create blinding commitments or establish a secure SSL session.

8. CONCLUSION

This paper presents a systemic solution to the problem of low-entropy keys. We present a new threat model, in which a device generating cryptographic secrets may have one communication session with an *entropy authority* which an eavesdropper cannot observe. Under this threat model, we describe protocols for generating RSA and DSA keypairs that do not weaken keys for devices that have a strong entropy source, but that can considerably strengthen keys generated on low-entropy devices. Our key generation protocols incur tolerable slow-downs, even on a CPU-limited home router. The threat model and protocols presented herein offer a promising solution to the long-standing problem of weak cryptographic keys.

Acknowledgements

We thank David Wolinsky, Ewa Syta, and Zooko Wilcox-O'Hearn for their helpful comments. This material is based upon work supported by the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contracts No. N66001-11-C-4018 and N66001-11-C-4022. This work was also supported by NSF and a Google faculty award.

9. REFERENCES

[1] M. Bellare, Z. Brakerski, M. Naor, T. Ristenpart, G. Segev, H. Shacham, and S. Yilek. Hedged public-key encryption: How to protect against bad randomness. In *ASIACRYPT*, pages 232–249. Springer, 2009.

[2] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73, 1993.

[3] M. Blum. Coin flipping by telephone: a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

[4] J. Camenisch and M. Stadler. Proof systems for general statements about discrete logarithms. Technical Report 260, Dept. of Computer Science, ETH Zurich, March 1997.

[5] S. Chokhani and W. Ford. Internet X.509 public key infrastructure certificate policy and certification practices framework, Mar. 1999. RFC 2527.

[6] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? In *CRYPTO*, pages 424–441. Springer, 1998.

[7] CVE-2001-0950: ValiCert Enterprise Validation Authority uses insufficiently random data, Jan. 2001.

[8] CVE-2001-1141: PRNG in SSLeay and OpenSSL could be used by attackers to predict future pseudo-random numbers, July 2001.

[9] CVE-2001-1467: mkpasswd, as used by Red Hat Linux, seeds its random number generator with its process ID, Apr. 2001.

[10] CVE-2003-1376: WinZip uses weak random number generation for password protected ZIP files, Dec. 2003.

[11] CVE-2005-3087: SecureW2 TLS implementation uses weak random number generators during generation of the pre-master secret, Sept. 2005.

[12] CVE-2006-1378: PasswordSafe uses a weak random number generator, Mar. 2006.

[13] CVE-2006-1833: Intel RNG Driver in NetBSD may always generate the same random number, Apr. 2006.

[14] CVE-2007-2453: Random number feature in Linux kernel does not properly seed pools when there is no entropy, June 2007.

[15] CVE-2008-0141: WebPortal CMS generates predictable passwords containing only the time of day, Jan. 2008.

[16] CVE-2008-0166: OpenSSL on Debian-based operating systems uses a random number generator that generates predictable numbers, Jan. 2008.

[17] CVE-2008-2108: GENERATE_SEED macro in php produces 24 bits of entropy and simplifies brute force attacks against the rand and mt_rand functions, May 2008.

[18] CVE-2008-5162: The arc4random function in FreeBSD does not have a proper entropy source for a short time period immediately after boot, Nov. 2008.

[19] CVE-2009-3238: Linux kernel produces insufficiently random numbers, Sept. 2009.

[20] CVE-2009-3278: QNAP uses rand library function to generate a certain recovery key, Sept. 2009.

[21] CVE-2011-3599: Crypt::DSA for Perl, when /dev/random is absent, uses the Data::Random module, Oct. 2011.

[22] dd-wrt. `http://dd-wrt.com/`.

[23] Electronic Frontier Foundation. The EFF SSL observatory. `https://www.eff.org/observatory`.

[24] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[25] P. Gallagher and C. Furlani. FIPS 186-3: Digital signature standard, 2009.

[26] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal–Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[27] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[28] G. H. Hardy and J. E. Littlewood. Some problems of *partitio numerorum*; III: On the expression of a number as a sum of primes. *Acta Mathematica*, 44(1):1–70, 1923.

[29] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, Aug. 2012.

[30] A. Juels and J. Guajardo. RSA key generation with verifiable randomness. In *PKC*, Feb. 2002.

[31] A. Klyubin. Some SecureRandom thoughts. `http://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html`, Aug. 2013.

[32] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. *IACR ePrint archive*, 64, 2012.

[33] littleblackbox: Database of private SSL/SSH keys for embedded devices. `https://code.google.com/p/littleblackbox/`.

[34] B. Möller. Algorithms for multi-exponentiation. In *Selected Areas in Cryptography*, pages 165–180, 2001.

[35] K. Mowery, M. Wei, D. Kohlbrenner, H. Shacham, and S. Swanson. Welcome to the Entropics: Boot-time entropy in embedded devices. In *IEEE Symposium on Security and Privacy*, 2013.

[36] NetBSD security advisory 2013-003: RNG bug may result in weak cryptographic keys. `ftp://ftp.netbsd.org/pub/NetBSD/security/advisories/NetBSD-SA2013-003.txt.asc`, Mar. 2013.

[37] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140. Springer, 1992.

[38] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[39] T. Ristenpart and S. Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[40] G. Taylor and G. Cox. Behind Intel's new random-number generator. *IEEE Spectrum*, Sept. 2011.

[41] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *IMC*, pages 15–27, Nov. 2009.

[42] A. Young and M. Yung. Kleptography: Using cryptography against cryptography. In *EUROCRYPT*, pages 62–74, 1997.