CA-in-a-Box

Mark Franklin, Kevin Mitcham, Sean Smith, Joshua Stabiner, and Omen Wild

Dartmouth PKI Lab and Department of Computer Science
Dartmouth College, Hanover NH 03755 USA
Contact authors: mark.franklin@dartmouth.edu, sws@cs.dartmouth.edu
http://www.dartmouth.edu/~pkilab/

Abstract. An enterprise (such as an institute of higher education) wishing to deploy PKI must choose between several options, all expensive and awkward. It might outsource certification to a third-party company; it might purchase CA software and appliances from a third-party company; it might try to build and maintain its own CA. In the latter two options, the enterprise faces the additional challenge of showing sufficiently safe practices to have its CA certified or cross-certified, for broader inter-operability.
This paper presents our research and development effort to address this problem. We use OpenCA to provide the basic functionality; we package it on a Linux installation on a bootable CD; we use the 1.1b TCG trusted platform module (standard on many desktop and laptop machines) to hold the private key; we also use the TPM to add assurance that the key can only be used when the system is correctly configured as the CA. This tool enables an enterprise to operate a CA possessing a degree of physical security and the ability to attest proper configuration to a remote certifier simply by booting a CD in a commodity machine. The code (and CD image) are all open-source, and will be available for free.

1 Introduction

Deploying PKI has many advantages for an enterprise. As members of a university, we are particularly receptive to (and practitioners of) standard PKI evangelism directed toward academic enterprises. Within the enterprise, PKI enables encryption and signing of e-mail and workflow documents, and identification and authorization for Web-based information services and network access. An Educase Net@EDU survey [12] shows several universities with production PKIs serving applications including virtual private networks, S/MIME e-mail, and document signing. With cross-certification (such as via the *Higher Education Bridge Certification Authority* [3]), these services can extend to permit applications such as resource sharing and document exchange between universities.

However, deploying PKI is one of the tougher and more expensive exercises a university IT department can endure. *Certification authorities (CAs)* are the backbone of most PKIs, but installing and maintaining them is notoriously complicated. The need to simplify the process of PKI setup has been apparent for several years and providing this service has become a profitable industry. Many universities have opted to outsource their CA to such corporations; this choice increases financial burden but limits headache and responsibility. (More than one university CIO has expressed frustration that certain

commercial CAs cannot cleanly project a priori the per-certificate cost the university will face.) Other universities operate CAs in-house, by purchasing or licensing software and cryptographic appliances and devoting IT man-hours to operating it. Still others try to reduce costs further by rolling their own CA from open source and commodity machines—but end up spending far more than expected in engineer and support staff time. The Educause survey cited above also showed that a university PKI's financial costs exceeded $50,000 per year—due to hardware, licensing, training, help-desk and upkeep.

These costs create a substantial barrier to adoption.

Our Project This challenge motivated our project. As a university, we began our PKI effort by exploring the various options and their costs, and elected to pursue the middle option: operating our own CA from commercial CA tools. However, the costs of this option were high—particularly since, in principle, a roll-your-own option with open source should have been easy and cheap.

As a side-project, we tried the open-source approach, and found it was neither easy nor cheap. It took two software engineers two weeks to get OpenCA installed and working. There are a lot of configurable options for OpenCA that require knowledge of the concepts and design of the OpenCA code, which are not trivial to come by. To make this path easier for others, we first attempted to make a cookbook: "Do X, then Y, then Z." We then realized we could automate it and get rid of many of the opportunities for mistakes.

In this paper, we attempt to lower the barrier to university PKI adoption, by producing tools that let a university set up a CA with hardware security and integrity protection simply by booting a CD. Furthermore, our tools provide the framework for remote attestation about the system's integrity, easing cross-certification.

The code and CD image will be available as open source.

This Paper Section 2 provides a brief background of certification authorities and their responsibilities. Section 3 presents the components we used to assemble our project; Section 4 presents the design and implementation of "CA-in-a-Box." Section 5 reviews related work. Section 6 discusses some directions for future research, and Section 7 concludes.

2 Certification Authorities

Although variants such as PGP and SPKI/SDSI have appeal, traditional X.509 PKI is the basis for the common PKI applications that motivate a university to adopt PKI. In this dominant paradigm, the enterprise depends on its certification authority to act as the intermediary between end users and relying parties (who may be other end users, or special entities such as the "course registration Web site").

In this standard framework, the CA acts as a trusted third party by using its private key to sign certificates. Each certificate binds a public key to information (typically identity) about the holder of the corresponding private key. The CA attests to the correctness of this binding; the university will have some mechanism in place, perhaps via a separate *registration authority (RA)*, to verify this binding.

Any certificate holder can present his certificate as confirmation that the University CA claims the certificate contains valid information. Anyone who trusts the CA as a certificate issuer for this domain should trust that his information applies to any entity that demonstrates knowledge of the certificate's private key.

Given the central role the CA plays in holding together this trust infrastructure, the primary responsibilities of a CA are to ensure its private key is only used for appropriately authorized operations—and to convince relying parties and other stakeholders that a significant barrier exists to keep it from being used for unauthorized ones.

Much of the associated work with setting up a university PKI stems from ensuring the CA fulfills these responsibilities. We consider some issues.

Failure Any system that relies heavily on its hardware must account for the possibility of hardware failure. CAs are no different in this case, particularly if the installation includes some type of special tamper-resistant hardware to hold the private key. The network environment might also be relevant. Some installations depend on the CA being online (in order to handle requests and post CRLs and such); others insist (to minimize the interfaces exposed to an adversary) that the CA remain isolated from the network.

One avenue of failure is *denial of service (DOS)*. A distributed DOS attack from another network (for online CAs) or a power failure can render the CA temporarily unavailable and cause user dissatisfaction. However, these failures are temporary and easily fixed. More critical are DOS failures that cause the destruction of the private root key such as a fire in the room containing the CA machine, or a defect in the secure hardware containing the key. (We have also heard anecdotes about vendors of secure hardware abandoning the product line and stranding their customers.) In these cases, the ability to sign certificates and add users to the PKI is lost. Additionally, if a CA root key protects a list of escrowed user encryption keys, we lose that data as well.

Alternatively, if the adversary *compromises* the CA's private key, consequences can cause chaos within a PKI. By learning the private key, the adversary has given himself the power to sign certificates as if he were the CA. The users of the PKI, therefore, can no longer trust the CA's signature. Each certificate signed by the private root key must be revoked and new certificates must be generated with a new key. The information leak causes a breakdown in the trust between users, as well as, between the CA and the PKI.

Human Element In addition to hardware failure, we must also consider the human element in certification. At some point in the certificate creation process, the CA must determine if it is going to vouch for the person requesting the certificate. The entire PKI trusts the CA (perhaps in conspiracy with an RA) to identify users correctly.

Consequently, the decision of how to architect the CA machinery holds great weight, as the details of the architecture may influence how easy or hard it is for an adversary—including a rogue insider—to cause invalid certificates to be issued. In a typical installation, even if we keep the private key inside a special device, the host machine determines when the private key is used and what data it operates on. Thus, the configuration of this machine becomes of paramount importance: Trojans, backdoors, unpatched software, or even extra accounts can subvert the system. The system administrator of the CA hardware may have unique power to sign or not to sign certain certificates and establish this trust between users.

Cross-Certification To enable cross-certification, the CA operator also needs to be able to establish various properties about CA operation to the satisfaction of the certifier. These properties include practices such as how (and how carefully) the CA verifies identities of the entities for which it is issuing certificates; separation of duties so no single individual can do bad things; physical and network security; what is logged and how the logs are kept; who has access and how access is controlled; how processes and procedures are defined and enforced; and where the private key is stored and how it is protected.

Design Goals This discussion leaves us with some design goals for our project.

- To keep equipment costs cheap, we need to use free software and commonly available commodity equipment.
- To ensure security of the CA private key when not in use, we need to exploit tamper-resistant hardware.
- To ensure security of the CA private key when in operation, we need to ensure that only a properly configured machine can request operations with this key.
- To assist in cross-certification, we need to make it possible for a remote CA to draw conclusions about the trustworthiness of this operation.
- To keep labor costs low, we need to make this easy to use.

3 Components

This section presents the components we used for this project.

3.1 OpenSSL

At its foundation, our CA needs to perform cryptographic operations.

OpenSSL is an open-source cryptographic library used by many standard applications that require cryptographic support [9]. OpenSSL is structured to permit use of underlying special-purpose cryptographic hardware; in OpenSSL, an *engine* is a module that enables OpenSSL to use some particular type of underlying hardware.

3.2 OpenCA

Our CA needs to carry out basic certification authority operations.

OpenCA is an open-source certification authority that uses OpenSSL [8]. We use OpenCA to manage the standard functionality of the CA: certificate generation, publication, revocation; we felt that OpenCA was the best choice to build upon as it is the most developed of the open source options. OpenCA supports publishing certificates and CRLs to LDAP. It provides an online RA component for handling certificate requests. Additionally, OpenCA is distributed with several Perl modules that can be used by other scripts we might choose to write ourselves [1].

3.3 Knoppix

Our CA needs to run on a properly configured system.

Proper configuration is necessary for basic operation. In theory, one can just install OpenCA and possess a CA. In practice, we found this process to be rife with subtle configuration issues. As discussed earlier, proper configuration is also necessary for secure operation.

We decided that a simple, easy-to-use way to ensure proper configuration would be to provide a properly configured system as a bootable CD. For this component, we chose *Knoppix*, which provides a customizable framework for putting complete Linux system (based on the Debian distribution) on a bootable CD image [6].

3.4 USB Flash Drive

Our CA needs space for installation-specific state. Minimally, we need space for static state, such as the private key (perhaps encrypted); however, a CA may accumulate dynamic state, such as logs of signed certificates.

To provide this, we configure the system to store its home directory on a removable USB flash drive.

3.5 TCPA/TCG TPM

It would increase security if our CA could store its private key in a safe place. However, we would like to avoid the expense and awkwardness of special-purpose equipment.

For this component, we chose the *Trusted Platform Module (TPM)*. Specified by the *Trusted Computing Group (TCG)* (formerly the *Trusted Computing Platform Alliance, TCPA*), the TPM is a smart-card like chip that is attached to the motherboard of many commodity PCs. We worked with the 1.1b version of the TPM [13], since that already comes by default with many desktops and laptops from IBM, and so is already somewhat ubiquitous.

The TPM acts as a credential store, keyed to its *platform configuration registers (PCRs)*. The host machine can store a value in the TPM and key it to specified values in a specified subset of the PCRs. The TPM will then decrypt and release that credential only when those PCRs have those values. Additionally, if the credential is an RSA private key, the host can request the additional feature of having the TPM never actually release the key—but rather only *use* it internally, and only when the PCR conditions are satisfied. The PCRs themselves are set in an interleaved way, starting with the boot ROM and BIOS, in order to reflect the configuration of that machine.

3.6 Bear/Enforcer

For holding the CA's private key in a TPM to be effective, we also need to take steps to tie it to the correct configuration for the CA on that machine. Commercial support for the TPM is still scarce, at this point; Linux support for the official *TCG Software Suite (TSS)* has recently been announced, but was not available. Furthermore, besides talking to the TPM, we need to figure out how to express "secure configuration" as a suite of

PCR values—which may be complicated by issues such as security-related software updates, which change the configuration but are necessary for the CA to remain secure.

For this component, we chose our lab's *Bear/Enforcer* code [7]. Bear/Enforcer is an open-source Linux tool suite that works with the 1.1b TPM to bind credentials to dynamic system configuration. Bear/Enforcer divides data into three important categories based lifespan. Long term data, like BIOS, the kernel, and Bear/Enforcer itself, are protected by the TPM-witnessed boot process. Medium term data, applications and daemons, are protected via a database of hashes. At initialization time, Bear/Enforcer checks that this database is current and properly signed by a potentially remote *security administrator*; at run time, a *Linux Security Module (LSM)* checks these hashes whenever an inode is touched. Shorter term data, like configuration files, live on a loopback filesystem[1] that can be encrypted and unmounted when not in use.

4 Design and Implementation

This section discusses how we put the above components together to solve our problem. Figure 1 sketches this design.

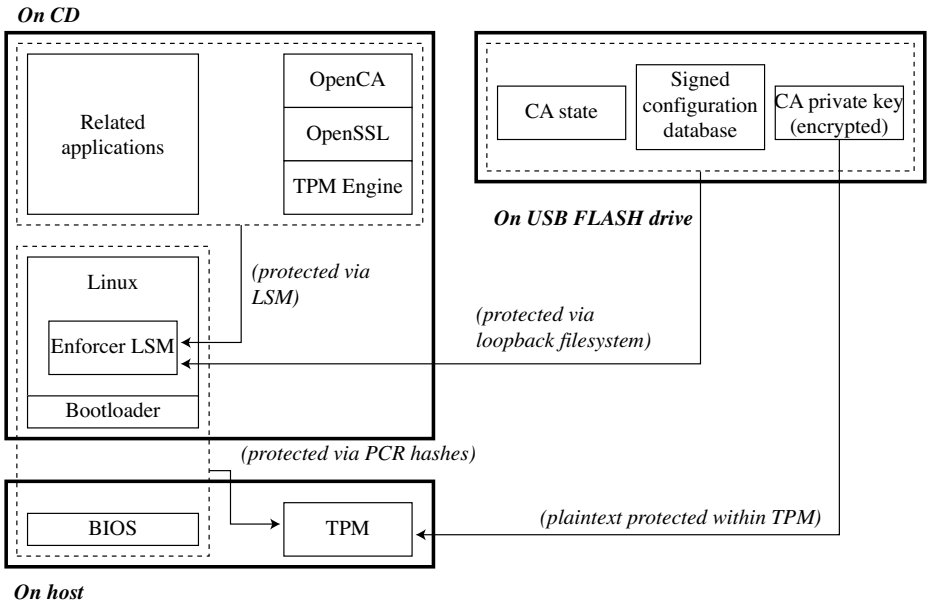

Fig. 1. A sketch of the system architecture for "CA-in-a-Box."

[1] This is a single file that the kernel will mount and treat as if it were a filesystem.

4.1 Putting the Pieces Together

Knoppix First, we remaster the Knoppix CD image by removing all unnecessary packages and features. This task both makes it easier to use, as well as decreases the *trusted computing base (TCB)*—e.g., if the kernel has no wireless extensions, then the CA cannot be compromised via a wireless attack.

OpenCA Then, we need to automate the configuration and use of OpenCA. OpenCA works by having two or three bases of trust: the client enrollment base, the registration authority, and the actual CA.

Different institutions have different enrollment needs and strategies. For some, security is paramount, and this goal tends to drive these institutions to an *offline CA*. The design idea is that the actual CA lives offline and only communicates with the world via "sneakernet." The enrollment tool sends requests to the RA, who makes some decisions about them. If approved, the RA sends the data up to the CA. The CA actually issues the certificate and sends it back to the RA; the RA delivers the certificate to the client. This offline approach is the use model OpenCA developers had in mind and is how OpenCA works most naturally.

For other institutions, however, considerations such as convenience, minimization of administrator overhead, and immediate fulfillment of certificate requests tip the balance in favor of *online CAs*.

We set out to adapt OpenCA to work in an online, immediate fulfillment with no administrator intervention required mode but were only able to get part way towards this goal. We combined the RA and CA functions onto one network accessible system and combined the RA and CA databases to reduce the number of adminstrator steps needed to move certificates through the system (when RA and CA are separate, these steps are necessary).

Unfortunately, we found that the architecture of OpenCA prevented us from being able to make online enrollment totally automatic without undue engineering effort. For some applications (or certificate assurance levels), requiring administrator intervention for each enrollment is still desirable, but the number of clicks involved in our implementation is higher than would be possible in a fully refactored implementation, and for many mass end user certificate deployments it would be useful to have an enrollment option that authenticates the end user and perhaps an RA "approver" and then totally automatically finishes the enrollment.

To set this all up, one must manipulate a large configuration file, that specifies all behaviors and allows options such as changing the number of layers (e.g., add extra RA steps) or having multiple RAs. This configuration file was large, and documentation was sparse. We eventually figured out how to modify it to have an RA and CA running on the same machine without conflicting; the RA and CA communicate via file copying. This model best suited our installation, and we also felt that universities looking for low-barrier way to adopt PKI would not want the hassle of multiple machines and "sneakernet."

Private Key Security One series of steps involves using the TPM to shelter the CA private key, to have the CA call the TPM for operations with this key. As discussed

earlier, OpenCA uses OpenSSL, and OpenSSL uses "engine" modules for implementations of cryptographic operations. The default engine (`openssl`) provides software functionality for OpenSSL cryptographic methods such as RSA, DSA, and RAND. The benefits of having this structure is the ease at which we can add new engines. When the user wishes to perform cryptographic operations on some specialized hardware device, he can simply load that engine into OpenSSL and then use the OpenSSL command-line normally. The loaded Engine has knowledge of the hardware device and how to interact with it [5].

To enable easy use of a TPM-housed RSA private key by OpenSSL, we built a `TPM Engine` module. We then built a custom compile of OpenSSL that contains this engine, and OpenCA to use the Engine API to look for the TPM.

We used the IBM utilities to generate keys manually in the TPM and then the engine takes over operations from there.

Configuration Security We also need to set up Bear/Enforcer to ensure that the TPM only uses the private key when the system is properly configured.

To initialize, the operator first boots the system and runs a script to "take ownership" of the TPM. The operator then inserts the CD and FLASH drive and reboots the system. The BIOS, the boot-loader on CD-ROM, the kernel, and the OpenCA configuration all get hashed into the PCRs; the filesystem is initialized on the external device. If running in "local" mode, the operator can run a script here to generate the Enforcer database and sign it; if running in a scenario where a remote party specifies secure configurations, we check the validity of the database this party has signed. We generate the symmetric key for the encrypted loopback filesystem and set that up, and store the key as a credential bound to this PCR suite. The operator then runs our OpenCA configuration scripts, which stores its state in the loopback, and generates the CA private key within the TPM itself, bound to this PCR suite. The CA config files along with public keys are stored on this removeable device in the loopback filesystem. This allows us to keep that information encrypted, on removable media, whenever it's not in use.

In normal boot, the BIOS, boot-loader and kernel are hashed into the PCRs. The OS gets loaded into system RAM. If Enforcer and the TPM determine the system configuration are satisfactory, the encrypted loopback filesystem is mounted so the CA configuration can be retrieved. OpenCA initializes and Enforcer checks the OpenCA binary. OpenCA tells openSSL to use the TPM Engine; if the configuration is still satisfactory, the encrypted private key is loaded into the TPM, which will then provide private key services to the CA.

Subsequent operation requires the CD, the FLASH drive, and that host machine.

4.2 Analysis

Security By reducing the amount of hardware being used we are able to more directly protect the hardware we do use. Because our CA does not require disk access, we do not need to worry about viruses or Trojans the CA machine might have picked up while performing non-CA duties; furthermore, the use of the bootable CD simplifies the problem of trying to maintain a special-purpose clean installation on that machine.

Denial of service can be a possibility if the TPM is destroyed or if hardware alterations fundamentally change the PCR values established during boot; Section 6 considers that further.

Protecting against a rogue system administrator is never an easy task. However, our approach provides us with an easy way to implement this protection. There are several components necessary for the "CA-in-a-Box" to work: a secret to help unlock the TPM's storage root key, the private key of the signer of the Enforcer database, and the dongle that contains the removable storage. We can distribute these elements among multiple people; potentially, we might distribute the database signer's key a remote site (see Section 6).

There are several parts of OpenCA that an adversary can exploit. OpenCA depends on OpenRA to manage certificate requests. It needs a MySQL database to store completed certificates pre-signing. It uses Perl to process every request. OpenSSL is essential to signing every certificate. Even the Web browser plays a key role by acting as the user interface to the CA.

Our Enforcer/TPM integration adds several additional layers of protection. If the adversary discreetly replaced any of these binaries with modified versions, he could easily trick the CA into signing a certificate that the adversary generated. However, when Enforcer is active, this is not possible: the Enforcer is set up to monitor each of these programs and cause a kernel panic (and tell the TPM to render the private key unusable) when any of them changes in a way not permitted by the signed configuration file.

Suppose an adversary gains access to our CA and inserts an unsigned certificate into the MySQL database. Then the adversary modifies some OpenCA Perl scripts so the next certificate exported from the database to be signed is his. As soon as the administrator loads OpenCA and the scripts are touched, Enforcer will detect the change and cause a kernel panic, shutting down the CA and stopping the attack. Likewise, if an adversary tried the same attack by modifying the MySQL binary the same response occurs. Enforcer provides an extra layer of protection not previously part of any open source CA.

Scalability The total number of users this system might support would be constrained by the database used to store the information, and the CA operator's time necessary to enroll them. If we assume one-year certificate lifetimes, and that a trained operator can reliably process an enrollment in five minutes, we project the system could get about 10,000 users in circulation (with 20 hours/week of operator time). OpenCA enrollment takes a lot of clicking, however; a more realistic projection might 1000 users/certs as it stands now. Processing twenty requests a week is enough to keep things moving, but not overwhelm the operator.

Streamlining and batching the enrollment process is an area for future work. E.g., here at Dartmouth College, we issue students certificates when they first matriculate. However, these newly matriculated students go through many other physical processes where their identities have been validated and they are batched together in a room—perhaps even after they have been issued College ID cards with RFID chips. Considerable potential for streamlining exists; we plan to explore this in future work.

5 Related Work

Jeff Schiller at MIT suggested building an offline CA from a laptop and Dallas iButton [10]. In addition to OpenCA, other open-source CA options include *XCA*, a graphical front-end to OpenSSL [4]; *pyCA*, not currently in active development [11]; and *Papyrus*, based on PHP [2].

Other experimental CA projects include COCA [15] and MOCA [14],

6 Future Work

In future work, we plan both to finish some necessary features, as well as integrate and test new ones.

In the former category, we need to design and implement a way to back up the CA configuration for a second machine, should changes to the initial host render it unusable. We should be able to accomplish this task with a fairly straightforward application of the TCG design of exporting one TPM's secrets to be used by a second designated back-up machine, perhaps in combination with secret-sharing among trustees. We also need to examine the failure scenario of the USB token being removed before our code unmounts it; plaintext data may remain there. For this problem, we may decrypt into RAM instead. (This should not be a significant security issue, however, since the primary secret—the CA private key—is protected by the TPM; what matters for the loopback filesystem is integrity.) We also want to stay abreast of ongoing work in our Bear/Enforcer project—such as for ensuring freshness of signed configuration files, and our recent integration of Enforcer with SE/Linux.

In the latter category, we want to finish building and testing tools to harness the configuration control and attestation features of Bear/Enforcer on the TPM in cross-certification. We plan to modify our Enforcer configuration-preparation tool for use by a bridge CA to establish signed databases for suitable CA configurations. We can then use the Bear/Enforcer attestation to communicate this status back to the bridge CA, thus easing enrollment in the bridge. We also plan to explore making this configuration information available to other relying parties, perhaps by setting up an attribute authority within Bear/Enforcer and having it sign attribute certificates about the CA configuration. We also plan to revisit the design decision to combine user enrollment, the RA, and the CA in one machine.

Eventually, we plan to validate these ideas in a broader pilot, perhaps in conjunction with HEBCA.

7 Conclusions

To conclude, our "CA-in-a-Box" project uses existing open source tools and commonly available commodity equipment to produce a CA that easy to install and use, but which also exploits hardware protections for the CA private key and software configuration. We offer this work to the community, in hopes that this helps promote broader use of PKI (at least by fellow universities) by easing the burden of establishing an enterprise PKI and having it cross-certified.

Acknowledgments and Availability

This work was supported in part by the Mellon Foundation, by the NSF (CCR-0209144), by Internet2/AT&T, by Sun, by Cisco, by Intel, and by the Office for Domestic Preparedness, U.S. Dept of Homeland Security (2000-DT-CX-K001). The views and conclusions do not necessarily represent those of the sponsors.

The authors are grateful to our many helpful colleagues—particularly here in the PKI Lab, and in the greater higher education PKI community—for their helpful suggestions and comments.

We are currently preparing our code for release. For more information, please contact `mark.franklin@dartmouth.edu`.

References

1. Chris Covell and Michael Bell. OpenCA Guides for 0.9.2+. `http://www.openca.org/openca/docs/online/`.
2. John Douglass. The Papyrus Project (Version 4), 2005. `http://www.cren.net/crenca/crencapages/papyrus.html`.
3. Higher Education Bridge Certification Authority. `http://www.educause.edu/hebca/`.
4. Christian Hohnstadt. XCA, 2003. `http://xca.sourceforge.net/`.
5. Pravir Chandra John Viega, Matt Messier. *Network Security with OpenSSL*. O'Reilly & Associates, Sebastopol, CA, 2002.
6. Knoppix linux. `http://www.knoppix.net/`.
7. J. Marchesini, S.W. Smith, O. Wild, A. Barsamian, and J. Stabiner. Open-Source Applications of TCPA Hardware. In *20th Annual Computer Security Applications Conference*. IEEE Computer Society, December 2004.
8. OpenCA PKI Development Project. `http://www.openca.org/openca/`.
9. OpenSSL: the Open Source toolkit for SSL/TLS. `http://www.openssl.org/`.
10. Personal communication.
11. pyCA–X.509 CA, 2003. `http://www.pyca.de/`.
12. Barry R Ribbeck. The PKI Working Group End User Deployment Matrix, 2004. `https://webspace.uth.tmc.edu/bribbeck/public/PKIWMATRIX.html`.
13. Trusted Computing Platform Alliance. Main Specification, Version 1.1b. `http://www.trustedcomputinggroup.org`, February 2002.
14. Seung Yi and Robin Kravets. MOCA:Mobile Certificate Authority for Wireless Ad Hoc Networks. In *2nd Annual PKI Research Workshop*, 2002.
15. Lidong Zhou, Fred B. Schneider, and Robert Van Renesse. COCA: A Secure Distributed Online Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.