HAND-HELD COMPUTERS CAN BE BETTER SMART CARDS

Dirk Balfanz and Edward W. Felten

Hand-Held Computers Can Be Better Smart Cards

Dirk Balfanz
Princeton University
balfanz@cs.princeton.edu

Edward W. Felten
Princeton University
felten@cs.princeton.edu

Abstract

Smart cards are convenient and secure. They protect sensitive information (e.g., private keys) from malicious applications. However, they do not protect the owner from abuse of the smart card: An application could for example cause a smart card to digitally sign any message, at any time, without the knowledge of the owner.

In this paper we suggest that small, hand-held computers can be used instead of smart cards. They can communicate with the user directly and therefore do not exhibit the above mentioned problem.

We have implemented smart card functionality for a 3COM PalmPilot. Our implementation is a PKCS#11 module that plugs into Netscape Communicator and takes about 5 seconds to sign an email message.

Generalizing from this experience, we argue that applications that are split between a PC and a hand-held device can be more secure. While such an application remains fast and convenient to use, it gains additional security assurances from the fact that part of it runs on a trusted device.

1 Introduction

Public key systems (like RSA [Rivest et al., 1978] or DSA [DSA, 1994]) promise to play a major role in the evolving global networked community. Since they solve the key distribution problem, they are especially useful for a big, open network like the Internet. In a public key system every individual possesses two keys: a *public* and a *private* key. The public key is published and known to everyone. The private key is only known to its owner, and kept secret from everyone else. Public keys are used to encrypt messages and verify signatures, and private keys are used to decrypt messages and produce signatures. If a thief steals your private key, he can not only read confidential messages sent to you, he can also impersonate you in electronic conversations such as email, World Wide Web connections, electronic commerce transactions, etc.

People should therefore protect their private keys adequately. One way to do so are *smart cards*. Smart cards are small[1], tamper-resistant devices that can be connected to a PC and store private keys. The PC cannot learn the private key stored on the smart card, it can only ask the card to perform certain cryptographic functions for which the private key is needed, such as calculating a digital signature or decrypting an email message. These functions are executed on the smart card. This way, the connected PC can be used to engage in electronic conversations on behalf of the smart card owner, but it does so without knowledge of the owner's private key.

Contrast this with a scenario in which the private key is stored on the PC itself (which is the most common case today). Although the key will probably be protected by a password, the PC itself could fall victim to an attack from the outside ([Gutmann, 1997]). If and when that happens, it is usually just a question of determination for the intruder to break the protection mechanism and learn the private key. Even worse, if the application that uses the private key turns out to be malicious (it could be infected by a virus, or turn out to be a tampered-with version of a well-known application, or some other Trojan horse), it can get to know the private key without any further ado.

[1]Usually, smart cards have the form factor of a credit card.

Smart cards protect users from security breaches of that kind. They are also quite convenient. Since they allow users to carry around their private keys with them, and at the same time connect to any PC, they allow users to engage in electronic conversations from any PC.

However, smart cards also have problems. Imagine, like in the scenario described above, that the PC to which the smart card is connected has been compromised, and that an email application that talks to the smart card is malicious. While the application will not be able to learn the private key on the smart card, it may do other things that are almost as bad. For example, when the user composes an email message to `president@whitehouse.gov` that reads

> I support you,

then the malicious emailer could, instead, send out an email that reads

> I conspired to kill you,

and could even have the smart card sign the latter. The digital signature might stand up in court and could get the owner of the private key into a lot of trouble.

The problem here is that the smart card does not have an interface to the user. A smart card is designed to protect the private key even if it is talking to a malicious application. However, it does not protect against *abuse* of the private key as shown in the example above. If the smart card had a user interface it could have shown – for verification purposes – the message it was asked to sign, and the user would have learned that the emailer was trying to frame him.

Recently, hand-held computers like the 3COM PalmPilot emerged in the marketplace. They are almost as small as smart cards, have superior computing power[2], and provide an interface to the user. In this paper we report on how we implemented smart card functionality for the 3COM PalmPilot (which we will simply call "Pilot" from now on). We describe how even in our initial attempt we achieved certain security properties that normal smart cards cannot deliver.

In Section 2 we give an overview of PKCS#11, which is a standard for "cryptographic tokens" (smart cards). In Section 3 we describe details of our implementation. In Section 4 we give an estimate as to how hard or easy it would be to extract the private key from a Pilot. Section 5 is reserved for a outlook into the future: What else can be done with hand-held computers? We summarize and conclude the paper in Section 6.

[2] With the exception of certain cryptographic operations - see Section 3.5.

2 Overview of PKCS#11

We have implemented a PKCS#11 library for Netscape Communicator. PKCS#11 [PKCS#11, 1997] is a "standard" drafted by RSA Data Security Inc. It defines sixty-odd prototypes for functions that together can be used to perform a wide range of cryptographic mechanisms, including digital signatures, public key ciphers, symmetric key ciphers, hash functions etc. The standard is designed with smart cards in mind[3]: The caller can have the PKCS#11 library perform certain functions, like producing a signature, without ever knowing the private key that is used in the process. In this section we will explain how PKCS#11 works, how Netscape Communicator uses it, and what the design of our PalmPilot implementation is.

PKCS#11 describes function prototypes and semantics for a *library*, which we will call *PKCS#11* or *cryptoki library*. An application can bind to a cryptoki library and call into its functions to perform certain cryptographic operations. The PKCS#11 world is populated by *objects*, which are sequences of attribute-value pairs. For example, the following is an object:

Attribute	**Value**
OBJECT_CLASS	PRIVATE_KEY
KEY_TYPE	RSA
TOKEN	YES
EXTRACTABLE	NO
LABEL	BOBS_PRIVATE_KEY
PRIVATE_EXPONENT	8124564...
MODULUS	7234035054...
⋮	⋮

Figure 1: A private RSA key

There are five different classes of objects: certificates, public keys, private keys, secret (i.e., symmetric) keys, and "data" objects. Objects can be created temporarily (e.g., a session key for an SSL connection), or can be

[3] It is also called the *cryptoki*, or cryptographic token interface, standard.

long-lived, in which case they are assumed to be stored on a "cryptographic token". The example above describes a private RSA key. It is stored on a smart card and marked "unextractable". This means that calls into the cryptoki library that try to read the values of various attributes of this key will fail. Hence, there is no way for the application to learn the value of this secret key (assuming that in fact is is stored on a smart card and not in the same address space as the application).

How can the application use this key to sign a message? The cryptoki library returns *handles* to the application, which are usually small integers and uniquely identify objects. Although the application will not be able to learn the private exponent of the key above, it can for example ask the cryptoki library if it knows of an object with the label "BOBS_PRIVATE_KEY". If such an object is found, a handle H is returned to the application, which can then ask the cryptoki library to sign a certain message with the object identified by H. Using the information known to it, the cryptoki library can sign the message and return the signature to the application. If the object is stored on the smart card, and the handle, message and signature are the only things exchanged between smart card and application, then the private key is safe. Object search and signature calculation have to happen on the smart card in this case.

PKCS#11 defines many more functions besides object search and message signing. The application can learn which cryptographic mechanisms are supported by the library, create and destroy objects, encrypt and decrypt messages, create keys, and more.

The cryptoki library will not perform certain functions (like signing a message) unless the user is *logged on* to the cryptographic token. Usually, the user has to provide a PIN or password to log on to the token. PKCS#11 distinguishes between tokens with or without a *trusted authentication path*. A token with a trusted authentication path is a smart card to which the user can log on directly, e.g., by means of a numeric key pad on the card. If the card does not have a trusted authentication path, then the application on the PC has to gather the PIN or password from the user, and send it to the smart card. We note that this is less secure than a trusted authentication path: The application can learn the PIN or password needed to "unlock" the private key on the smart card. Smart cards usually do not have a trusted authentication path.

Netscape Communicator supports PKCS#11, which means that smart card vendors can provide a cryptoki shared library [PKCS#11, 1998]. Communicator will then bind to that library and use it for certain cryptographic operations. In particular, Communicator uses the library for key pair generation, S/MIME encrypted email [Dusse et al., 1998a, Dusse et al., 1998b] and client authentication in SSL connections [Freier et al., 1996]. For S/MIME the cryptoki library has to support RSA [Rivest et al., 1978], DES [DES, 1993], RC2 [Rivest, 1998], and Triple-DES [Schneier, 1996]. Communicator only supports RSA key pair generation, so client authentication – although technically feasible with any signature scheme – is done using RSA signatures.

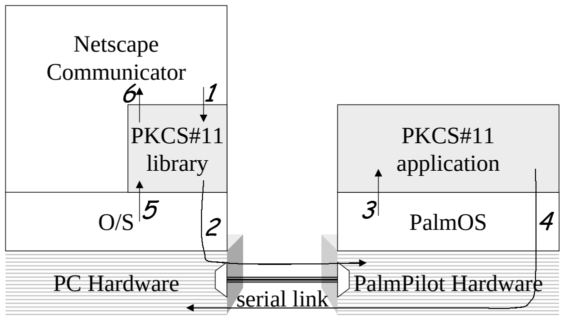

Figure 2: Overview of our PKCS#11 implementation

We implemented a cryptoki library that supports the above mentioned ciphers and signature schemes. The shared library that plugs into Communicator only serves as a dispatcher of requests to the Pilot. For the Pilot, we have written an application that receives those requests, performs the necessary functions and sends the results back to the PC.

Figure 2 shows how our implementation works. The PC and the PalmPilot are connected through a serial link. We implemented the pieces shown in grey: a plug-in library for Netscape Communicator and an application for the PalmPilot. Communicator calls into our library when some cryptographic operation needs to be performed (1). If the request cannot be handled by the library itself, it forwards it to the PalmPilot (2). On the PalmPilot, the operating system notifies our PKCS#11 application when a request arrives (3). The application processes the request and returns the result to the PC (4), where it is received by the cryptoki library (5). The library will then return this result or a result based on what it received from the PalmPilot back to Communicator (6).

It is worth pointing out what the trusted components are in this model: We trust that the PKCS#11 application on the PalmPilot is not tampered with and performs correctly. We trust in the same way the operating system on the PalmPilot and its hardware. On the other hand, we

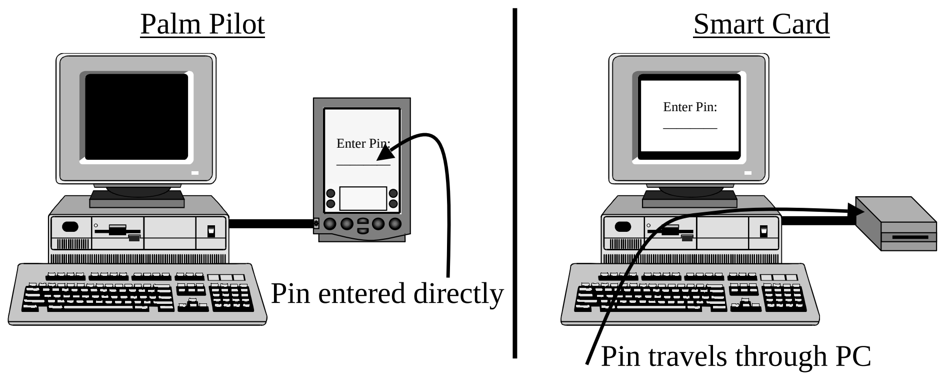

Figure 3: Information flow for PIN, contrasting traditional smart card and PalmPilot.

do not have to trust Communicator, the operating system on the PC or its hardware to collaborate with us. We do not even trust that the PKCS#11 library does what it is supposed to do. The PKCS#11 application on the Pilot is written defensively and works[4] even in the case where the PKCS#11 library is replaced by malicious code.

3 The Implementation

Our implementation runs on Windows 95, Windows NT, and Linux for the Communicator plug-in, and on a 3COM PalmPilot Professional for the Pilot side. It practically turns the Pilot into a smart card. For the cryptographic functions we used the PalmPilot port of SSLeay [Young et al., 1998]. In the following section we will highlight a few points of our particular implementation.

3.1 Key Pair Generation

To create an RSA key pair, we need a good source of randomness. In a traditional smartcard, if the attacker knows the random number generating algorithm and initial seed, we cannot hide the private key from him. He can simply perform the same computation as the smart card. PKCS#11 provides functions to re-seed the random number generator on the card, but the application never has to call them.

On the Pilot, we can use external events to provide randomness that the application cannot observe. When the Pilot generates a new key pair, we ask the user to scribble randomly on the screen of the Pilot, thus providing random bits we use for seeding the random number generator.

[4]"Works" means that it does not leak any information it is not supposed to leak. It does not mean that the system does not crash, for example.

3.2 Logging on to the Pilot

The Pilot is a cryptographic token with a trusted authentication path. Therefore, the user does not enter his password through Communicator, but directly into the Pilot. This way Communicator cannot learn the PIN or password needed to unlock the private keys. If it knew the PIN or password, it could try to use the Pilot without the user's knowledge or - even worse - make the PIN known to adversaries who might later find an opportunity to steal the Pilot from its owner. Figure 3 shows the difference in information flow between a traditional smart card and our PalmPilot implementation.

3.3 Signing Email Messages

In order to send a signed email message, all the user needs to do is log on to the Pilot and put it into its cradle, which will connect it to Communicator. Communicator will ask the Pilot to sign the message and also retrieve a certificate stored on the Pilot (unlike private keys, certificates are extractable objects and can therefore be retrieved by the application). Then, Communicator adds the signature and the certificate to the message according to the S/MIME standard, and sends it off.

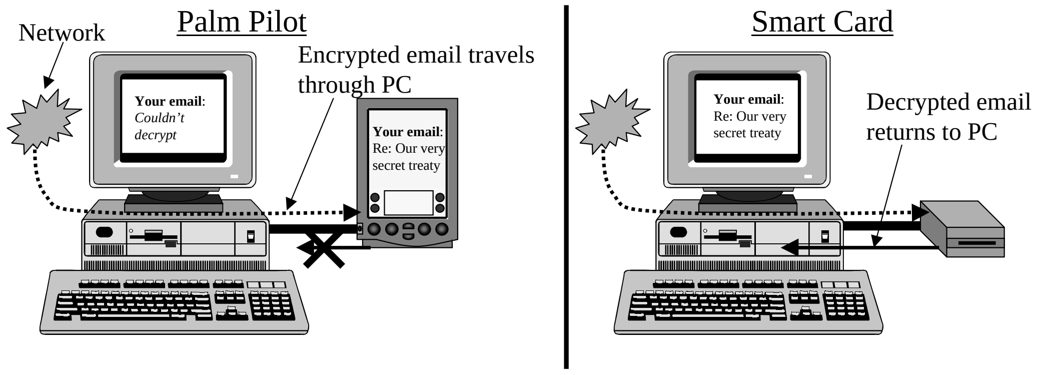
Palm Pilot
Network
Encrypted email travels through PC
Your email: Couldn't decrypt
Your email: Re: Our very secret treaty
Smart Card
Your email: Re: Our very secret treaty
Decrypted email returns to PC

Figure 4: Information flow for reading encrypted email, contrasting traditional smart card and PalmPilot.

3.4 Receiving Encrypted Email

When Communicator receives a message that has been encrypted with the user's public key, it will first ask the Pilot to use the corresponding private key to unwrap the symmetric (RC2 or DES) key used to encrypt the contents of the message. Depending on the preference setting on the Pilot, it will either signal an error, or unwrap the symmetric key and store it as an unextractable object on the Pilot. In the former case, Communicator will ask to "decrypt" a certain string which happens to be the wrapped key, and hence obtain the symmetric key needed to decrypt the message.[5] In the latter case, Communicator will send the encrypted message to the Pilot and ask it to decrypt it using the symmetric key just unwrapped. The Pilot will decrypt the message, display it on its screen, and send bogus information back to Communicator. So, depending on the preference settings on the Pilot, Communicator may or may not be able to see the decrypted contents of a message[6].

Users that have little trust in the PC that their Pilot is connected to can use this feature to read encrypted email on their Pilot without risking its contents to become known to anyone. This cannot be done with traditional smart cards since they lack a user interface. See Figure 4 to illustrate this point.

[5]Note the difference between "unwrappinng" and "decrypting". An unwrapped key is not returned to the caller (only a reference to it), while a decrypted key is.

[6]If the Pilot agrees to unwrap the key, it will refuse to "decrypt" it, so that even a malicious version of Communicator would not be able to decrypt the message on its own.

3.5 Performance Issues

A Pilot is not as fast as a smart card when it comes to long integer arithmetic. Signing/decrypting a message with a 512 bit key (the key size in exportable versions of Communicator) takes about 5 seconds. Signing/decrypting with a 1024 bit key takes about 25 seconds. (These measurements were made with version 2.01 of pilotSSLeay.)

Creating a 512 bit RSA key pair takes a couple of minutes. Creating a 1024 bit key pair takes 30 minutes. These numbers may vary since a randomized algorithm is used to find the primes.

Communicator often calls into the cryptoki library to reassure that the token is still connected, to exchange information, to search for objects, etc. Through a rather slow serial link, this conversation takes up a lot of time. We built in Communicator-side caches for most of the information that Communicator learns from the Pilot. So what users experience is a flurry of messages going back and forth between PC and Pilot when they start using it during a session. However, after a while the Pilot will only be contacted to actually perform a cryptographic operation, and the time experienced by the users closes in on the numbers given above.

4 Protecting Sensitive Data

How safe is the user's sensitive data if the Pilot gets into the hands of adversaries? In our implementation, sensitive parts of the private key are stored encrypted in nonvolatile RAM. We derive a DES key from the PIN or password that the owner uses for logging on to the Pilot and use it to encrypt the sensitive parts of the private key. Later, they are only decrypted just before they are used, and erased after they are used. When the user logs off, the PIN or password and the derived DES key are erased.

It is very easy for an adversary to read the encrypted key out of non-volatile RAM. He then needs to perform an attack on the DES encryption, or alternatively a dictionary attack on the PIN or password. Since only the actual bits of sensitive data (which are unknown to the attacker) are encrypted, a straightforward known plaintext attack is not possible. Instead, the attacker would have to perform an expensive multiplication to test whether he had found the correct private key. We therefore assume that the private key is reasonably safe, with the weakest link in the chain probably being the PIN or password, which could be too short or part of a dictionary.

For a device like a Pilot, a different threat is more imminent. Our PKCS#11 application usually shares the Pilot with many other applications. Since there is no memory protection, other applications might be able to read the decrypted private key if they manage to interrupt the PKCS#11 application just at the right time. So, for better security users should be very careful about what kind of applications they install on their Pilot.

To alleviate this situation, the PalmPilot would need a comprehensive security architecture. First, the operating system should enforce memory protection and add access control to its resources such as the databases in non-volatile RAM. Second, downloading software onto the PalmPilot should be restricted; we could for example imagine a password needed to download new software onto the Pilot. Third, downloaded software should be put in sandboxes that are enforced by the operating system. There should be a way to relax sandboxes, perhaps based on digital signatures, in order to give certain applications access to more functionality. With a system like this, a user could for example download a game from an unknown source and place it into a tight sandbox from where it cannot possibly learn things about the other applications running.

Since we started this work in 1997, a number of vendors and research projects have tried to address the problem of operating system security in hand-held computers. Recent versions of Microsoft's Windows CE operating system include a feature that allows only applications signed by designated principals to be considered "trusted". Untrusted applications are barred from calling a set of system calls considered sensitive. Sun and 3COM have recently announced that Sun's Java Platform 2, Micro Edition, will be available for the 3COM Palm series of devices. Also, since smart cards themselves have become more powerful, the operating systems on smart cards now tend take into account the scenario of multiple, mutually suspicious applications running on the same card. The techniques used on modern smart cards (secure file systems, etc.) could be applied to hand-held computers.

Once this security architecture is in place, we can turn to physical security: Now that it is impossible for an adversary to inject software into the Pilot, we need to make sure that the data cannot be read out in some other way. To prevent this, the Pilot could be equipped with tamper-resistant hardware, which would make it difficult to obtain the data in question by physical means. Equipping a Pilot with tamper-resistant hardware is not too hard since it is bigger than a smart card and there is more space to accommodate tamper-aware protection mechanisms. But as long as there is an easy way to install arbitrary applications on the Pilot, the additional cost of a tamper-resistant hand-held computer would not be justified.

5 Future Directions

So far, we have described a particular system that implements smart card functionality on a PalmPilot. We have also seen that a hand-held computer has potential for better security, since it provides a direct user interface. The fundamental reason for the desirable security features is that the Pilot is more *trusted* than the PC. Just like a smart card, we always carry it around with us and are fairly certain that no-one tampered with it. We have a good overview of the software running on the Pilot and usually know where it came from. Contrast this with a PC, which may easily run hundreds of different processes at a time. These different processes may open up vulnerabilities to attack. Moreover, if we use our Pilot with an arbitrary PC, we do not know whether that PC has not been specifically prepared to undertake malicious acts.

In this section we are going to explore the possibilities of the following scenario: A trusted, small, moderately powerful computer working together with a more powerful, big, but untrusted, PC to achieve some functionality. The trusted small device is needed to assure a certain amount of security, and the big PC is needed to give a convenient and powerful interface for security-insensitive operations. For example, the PC could display media-rich documents that are not sensitive. This is really just a generalization of the case discussed so far in this paper.

As a first example, let us get back to email. Our implementation presented in Section 3 has at least two shortcomings:

1. The user does not see, on the Pilot, the message that is to be digitally signed[7]. This means that a malicious version of Communicator could forge email.

2. Only for encrypted messages can the user decide whether or not they should be displayed on the PC.

A better approach would be if the email application was controlled from the Pilot. The Pilot establishes an encrypted connection – through the PC it is connected to – to the user's mail host. Then the user decides where a certain message should be displayed.

Another possible application is electronic commerce. Here, also, smart cards are often used. However, because of the lack of an interface, we do not really know what our smart card is doing and how much money it is spending. With a Pilot, the picture looks different: We again consider a scenario where three players are involved. First, there is a server offering goods to purchase. In addition, there is a trusted Pilot (or similar device) which is connected to an untrusted PC. The Pilot carries our electronic cash. On the PC, we launch the client application that connects to the server and displays the offered goods. The Pilot also authenticates itself to the server and establishes an encrypted channel that the PC cannot decrypt.

The client application can be used to view the items, get price information, initiate a purchase, etc. Before the Pilot spends any money, it displays relevant information such as the price and a description of the product on its screen. Only if and when the user confirms this *on the Pilot* can the transaction proceed. We note that even if the server and PC collaborate, the Pilot, which acts as an electronic "wallet", will not dispense more money than the user acknowledges.

Generalizing from these examples, we envision a new programming paradigm we call *Splitting Trust*. Under this paradigm, applications are split to run on different devices. Part of the application runs on a small, trusted device, and part of the application runs on a bigger, more powerful, but untrusted, device. We believe that this splitting enables users to get both security and computing power. They get security because a crucial part of their application runs on a trusted device. They get computing power because the more powerful device can be used to run non-crucial parts of the application. Part of our future work will be to provide middleware to enable easy splitting of applications in this fashion.

[7] Communicator never passes the message content into the cryptoki library. Rather, it calculates the hash itself and then just lets the cryptoki library sign the hash of the message.

6 Conclusions

In this paper we have argued that small hand-held computers can be used instead of smart cards. Moreover, they provide a direct interface to the user. We implemented a PKCS#11 library for Netscape Communicator and corresponding smart card functionality for a 3COM PalmPilot.

In our implementation, the PalmPilot provides a trusted authentication path and gives the user a choice where an encrypted email message should be displayed: in Communicator on the PC or on the PalmPilot itself. This increases the user's privacy above the level provided by traditional smart cards.

We also propose to generalize from our experience and to introduce a new programming paradigm. Under this new paradigm, applications are split into two parts: One part runs on a trusted, but small and only moderately powerful, device; the other part runs on a bigger, more powerful, but possibly untrusted, device like a PC in a public place. Splitting applications will give us both certain security assurances on one hand, and convenience and speed on the other hand. We plan to provide middleware to assist the process of splitting applications.

7 Related Work

In [Yee, 1994] Yee introduces the notion of a *secure coprocessor*. A secure coprocessor is a tamper-resistant module that is part of an otherwise not necessarily trusted PC. Certain goals (notably copy protection) can be achieved by *splitting* applications into a *critical* and *uncritical* part; the critical part runs on the secure coprocessor. While this idea is very similar to ours, the context is different: In the world of secure coprocessors the user is not necessarily trusted (the coprocessor secures information from, among others, the user using it). On the other hand, the user always trusts the secure coprocessor, even if it is part of an otherwise unknown PC. In our world, the user/owner of the PalmPilot is trusted by definition (the whole point of our design is to protect the user). The PC, or any of its parts (even if it looks like a secure coprocessor) is never trusted.

Gobioff et al. notice in [Gobioff et al., 1996] that smart cards lack certain security properties due to their lack of user I/O. They propose that smart cards be equipped with "additional I/O channels" such as LEDs or buttons to alleviate these shortcomings. Our design meets their vision, but we come from the opposite direction: We take a hand-held computer that already has user I/O and implement smart card functionality on it.

Boneh et al. implemented a electronic cash wallet on a PalmPilot, which is quite similar to what we describe in Section 5 [Boneh and Daswani, 1999].

Cryptographers have dealt with the "splitting trust" scenario for some time now, even though the work is often not presented from that perspective. For example, Blaze et. al [Blaze, 1996, Blaze et al., 1998] want to use a powerful PC in concunction with a smart card for symmetric key encryption because the PC provides higher encryption bandwidth. However, the PC is not trusted to learn the secret key. "Function hiding" work (e.g. [Sander and Tschudin, 1998]) is usually presented from a persective of copyright protection, but essentially it is also an instance of splitting trust. Boneh et al. use results from multi-party threshold cryptography to speed up RSA key pair generation on the PalmPilot with the help of an untrusted server, which participates in the key pair generation, yet will not learn the private key (see [Modadugu et al., 1999]).

8 Acknowledgments

We would like to thank Ian Goldberg for porting SSLeay to the PalmPilot, enhancing Copilot, and providing valuable Pilot programming tips. We also extend our thanks to Bob Relyea for help with some PKCS#11 details.

References

[Blaze, 1996] Blaze, M. (1996). High-bandwidth encryption with low-bandwidth smart cards. In *Proceedings of the Fast Software Encryption Workshop*, volume 1039 of *Lecture Notes in Computer Science*, pages 33 – 40. Springer Verlag.

[Blaze et al., 1998] Blaze, M., Feigenbaum, J., and Naor, M. (1998). A formal treatment of remotely keyed encryption. In *Proceedings of Eurocrypt '98*, volume 1403 of *Lecture Notes in Computer Science*, pages 251 – 265. Springer Verlag.

[Boneh and Daswani, 1999] Boneh, D. and Daswani, N. (1999). Experimenting with electronic commerce on the PalmPilot. In *Proceedings Eurocrypt '99*, volume 1648 of *Lecture Notes in Computer Science*, pages 1 – 16. Springer Verlag.

[DES, 1993] DES (1993). *Data Encryption Standard*. National Institute of Standards and Technology, U.S. Department of Commerce. NIST FIPS PUB 46-2.

[DSA, 1994] DSA (1994). *Digital Signature Standard*. National Institute of Standards and Technology, U.S. Department of Commerce. NIST FIPS PUB 186.

[Dusse et al., 1998a] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and Repka, L. (1998a). *S/MIME Version 2 Message Specification*. IETF - Network Working Group, The Internet Society. RFC 2311.

[Dusse et al., 1998b] Dusse, S., Hoffman, P., Ramsdell, B., and Weinstein, J. (1998b). *S/MIME Version 2 Certificate Handling*. IETF - Network Working Group, The Internet Society, RFC 2312 edition.

[Freier et al., 1996] Freier, A. O., Karlton, P., and Kocher, P. C. (1996). *The SSL Protocol Version 3.0*. IETF - Transport Layer Security Working Group, The Internet Society. Internet Draft (work in progress).

[Gobioff et al., 1996] Gobioff, H., Smith, S., Tygar, J. D., and Yee, B. (1996). Smart cards in hostile environments. In *Proceedings of The Second USENIX Workshop on Electronic Commerce*, Oakland, CA.

[Gutmann, 1997] Gutmann, P. (1997). How to recover private keys for microsoft internet explorer, internet information server, outlook express, and many others - or - where do your encryption keys want to go today? `http://www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt`.

[Modadugu et al., 1999] Modadugu, N., Boneh, D., and Kim, M. (1999). `http://theory.stanford.edu/~dabo/abstracts/RSAgenkey.html`.

[PKCS#11, 1997] PKCS#11 (1997). *PKCS#11: Cryptographic Token Interface Standard, Version 2.0*. RSA Laboratories.

[PKCS#11, 1998] PKCS#11 (1998). *Implementing PKCS#11 for the Netscape Security Library*. Netscape Communications Corporation, Mountain View, California. `http://developer.netscape.com:80/docs/manuals/security/pkcs/pkcs.htm`.

[Rivest, 1998] Rivest, R. (1998). *A Description of the RC2(R) Encryption Algorithm*. IETF - Network Working Group, The Internet Society. RFC 2268.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Sander and Tschudin, 1998] Sander, T. and Tschudin, C. (1998). On software protection via function hiding. In *In Proceedings of the Second Workshop on Information Hiding*, Lecture Notes in Computer Science. Springer Verlag.

[Schneier, 1996] Schneier, B. (1996). *Applied Cryptography*, chapter 15.2 Triple Encryption, pages 358–363. John Wiley.

[Yee, 1994] Yee, B. S. (1994). *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University.

[Young et al., 1998] Young, E. et al. (1998). SSLeay and SSLapps. `http://psych.psy.uq.oz.au/~ftp/Crypto/`.