Experimenting with Electronic Commerce on the PalmPilot

Neil Daswani[1] and Dan Boneh[2]

[1] Stanford University, Stanford CA 94305, USA,
daswani@cs.stanford.edu,
WWW home page: http://www.stanford.edu/~daswani/
[2] Stanford University, Stanford CA 94305, USA,
dabo@cs.stanford.edu,
WWW home page: http://www.stanford.edu/~dabo/

Abstract. This paper describes our experience with implementing an electronic payment system for the PalmPilot. Although Palm OS lacks support for many desired security features, we are able to build a system suitable for small payments. We discuss the advantages and disadvantages of using a PDA to make secure payments as opposed to using a smartcard or a desktop PC. In addition, we describe the engineering of PDA-PayWord, our implementation of a commerce protocol that takes advantage of both elliptic curve and RSA public key cryptography to support payments efficiently on PDAs with limited processing capability.

Keywords: electronic commerce, personal digital assistants, PalmPilot, digital wallet, electronic payment systems.

1 Introduction

The explosive growth of the market for Personal Digital Assistants (PDA's) has led to a wealth of new applications for them. In this paper, we experiment with electronic commerce for PDA's. Our motivation is clear: since consumers are already carrying digital assistants, why not use them for payments? For example, one could walk up to a vending machine, connect the PDA to the machine via the infrared link and make a purchase. One application of our system does just that (see Section 4.2). One may wonder whether electronic commerce on a PDA is any different than electronic commerce using other digital devices such as smartcards or desktops. The answer is simple – PDA's are a middle ground between smartcards and desktops. They posses advantages and disadvantages over both. As a result, payment schemes and electronic commerce applications need to be fine tuned to properly execute on a PDA.

When compared with smartcards, PDA's appear to be at a disadvantage: (1) they are not *tamper resistant*, (2) unlike many smartcards they are not equipped with cryptographic accelerators, and (3) they do not have limited lifetime as many smartcards do. These features (especially tamper resistance) are helpful in simplifying real world payment schemes. On the other hand, PDA's have several advantages over common smartcards. First and foremost, they have a direct line of communication with the user. Common smartcards can only interact with the user through an *untrusted* reader. In addition, unlike smartcards, PDA's have a reasonable (yet limited) amount of non-volatile memory[1]. Consequently, they can store a longer transaction log. PDA's are also better at general purpose computations; for example, computations that take place during protocol negotiation. We note that when comparing smartcards and PDA's, the issue of unit cost is irrelevant – PDA's may become just as widely deployed as smartcards.

When compared with desktops, PDA's appear to be at a disadvantage once again. Due to their limited memory capacity, they can only store a limited amount of financial instrument data, and a limited number of transaction certificates. Similarly, due to their limited computing power they cannot engage in complicated cryptographic protocols. An intensive protocol such as SET [23] may take too long to execute on a PDA. On the other hand, PDA's are portable and can be used in many environments where a desktop is not available.

The above discussion explains why many existing payment systems cannot simply be ported to a PDA. One must design the system keeping in mind both the limited security features and the limited computation resources. In this paper, we describe our experience with building a payment system for 3Com's popular PalmPilot [1]. Our system is designed to be portable, and the lessons we learned apply to other PDA's, e.g. ones based on Windows CE. Our choice for using the PalmPilot is due its current dominance of the PDA market. We note that existing financial applications [9] for the PalmPilot enable one to keep track of spending, but none provide a payment scheme.

To implement an electronic commerce system one must start with a basic framework for making payments. Our starting point is a generic wallet architecture that we had previously designed [7]. We briefly review the wallet architecture in Section 1.2. We stress that the wallet only provides the skeleton for this work. Our main focus is the design of payment components that can be adequately used on a PDA. We describe these in Section 3. The system implementation details are presented in Section 4, where we also describe our first vendor: the Pony Vending Machine. We begin by describing the (lack of) security features on the PalmPilot.

1.1 Existing PalmPilot Security Features

All data on the PalmPilot is stored in either ROM or RAM. The PalmPilot does not have a disk drive or any other form of persistent memory. The RAM on the PalmPilot is divided into *dynamic* and *storage* memory. Dynamic memory is used

[1] The PalmPilot IIIx comes with four megabytes of RAM. A Windows CE machine typically has four to eight Megabytes. We note that most memory on a PDA is devoted to applications such as an appointment book. A commerce application can only use a small portion of the memory on the device.

as working space. It holds the program stack and other short term data. On the other hand, storage memory is non-volatile and plays the role of a disk drive on desktops. *Databases* in the storage RAM are the equivalent of files on a disk drive. A database on the PalmPilot is made up of a sequence of records. Database access on the PalmPilot is somewhat different than a traditional operating system. In traditional systems, a file is read into memory before data stored in it can be accessed or modified. On the PalmPilot, database records are always in memory. There is no need to move them into dynamic storage to operate on them. They can be *edited in place*.

Unfortunately, Palm OS provides very little support for access control. Although databases have a *creator ID* associated with them, the operating system does not prevent an application from opening a database that does not belong to it. Consequently, malicious applications can easily tamper with sensitive data on the PalmPilot. Individual records in databases have a "secret" attribute associated with them. However, the core OS only views this attribute as a suggestion. There is nothing preventing an application from gaining access to records marked secret. Essentially, the secret attribute tells the application not to display these records unless the user types in a password. The application itself is not prevented from processing records marked secret. It is a bit surprising that although the OS provides a facility for requesting the user to enter a password, it does not provide any support for encrypting data using the password.

The lack of support for access control has prompted a number of developers to attempt to remedy the situation [22]. A number of shareware applications for the PalmPilot enable the user to store data in an encrypted database. This is especially useful when using the PalmPilot to store sensitive information such as PIN's and login passwords. Unfortunately, at the moment these applications use weak encryption. We note that Ian Goldberg [11] ported to the PalmPilot the cryptography-related parts of Eric Yung's SSLeay library. We make use of this port in our protocols.

To summarize, access control on the PalmPilot is poor. The operating system does not enforce access control and there is no support for privacy via encryption. This is very different than a typical smartcard operating system where there is a clear partitioning of memory between applications. We hope future versions of the PalmPilot will pay more attention to security issues. This is likely to be the case as indicated by the recent collaboration between Certicom and 3Com. Certicom is integrating its elliptic curve encryption technology into Palm OS. Finally, we note that things appear to be better on Windows CE, where some support for access control is provided by the operating system.

1.2 A Brief Review of SWAPEROO: The Simple Wallet Architecture for Payments, Exchanges, Refunds, and Other Operations

.

The starting point for our payment system is a generic wallet architecture that can accommodate multiple payment protocols and multiple financial instruments. The wallet provides the groundwork for us to build on. Since it is not the focus of this paper, we only give a high level overview. Details of the architecture are given in [7]. The overview is necessary to understand how the various payment components described in Section 2 and 3 fit together. The wallet design is based on four principles: (1) *extensible*, it is relatively easy to add new types of financial instruments and payment protocols; (2) *non-web-centric,* does not rely on web based protocols; (3) *symmetric*, we use similar software API's on both the user and vendor sides; (4) *client-driven*, all wallet operations are initiated by the client, not the vendor.

Both the user and vendor make use of the same wallet architecture. The user wallet is part of a larger user application. Similarly, the vendor wallet is part of a larger vendor application. Once the user and vendor applications agree on a certain payment operation (e.g. purchase item #3, or refund 5$) the wallet controllers on both sides take control to execute the transaction. The first thing they do is determine the common financial instruments they support. For example, the user wallet may contain CyberCash coins [6], Millicent scrips [17] and a Visa credit card. The vendor may only accept credit cards and CyberCash coins. The common instruments are the credit card and CyberCash coins, in which case, the user is asked to choose which instrument to use.

Once the financial instrument is chosen, the user and vendor wallet controllers negotiate the payment protocol. Indeed, multiple protocols can be used to make payments with a particular financial instrument. For example, when paying with a credit card one could use SET [23], although other protocols may be used as well. This protocol negotiation is transparent to the user. Once the two agree on a common protocol, the protocol is activated to execute the transaction.

To carry out the above interaction, the wallet architecture provides controllers for manipulating financial instruments and applying protocols to them. The wallet is made up of a Protocol Manager (PM), an Instrument Manager (IM), a Wallet Controller (WC), and a User Profile Manager (UPM). A Communication Manager handles low level communication. Once our wallet architecture was successfully ported to the PDA, we were free to concentrate on the design of financial instruments and payment protocols to be used.

2 Electronic Commerce for a PDA

If PDA's are to be used for digital payments, they will most likely be restricted to small payments (i.e. transactions under 10$). For larger payments one may wish to rely on stronger security properties. Our hope is that PDA's may be used in both local and remote payment transactions. For local transactions, a user may walk into a shop, place the PalmPilot in the vendor's cradle (or use the infrared port) and make a payment. For remote transactions, one may connect the PDA to the Internet (using a PDA modem or by using a desktop as a gateway). The two modes are different: for local transactions there is no need to encrypt communication on the wire. For remote transactions, one must first establish a secure link to the vendor. Throughout the discussion in this section we do not distinguish between the two modes. At the moment, our implementation only supports local transactions that do not require link encryption.

2.1 Performance of Cryptographic Primitives on the PalmPilot

We provide timing measurements for cryptographic primitives running on the PalmPilot[2]. These timing measurements help determine if a payment scheme is feasible for the PalmPilot or if it is too complex. The PalmPilot Professional runs on a Motorola DragonBall chip (68K family) at 16MHz. The figures below are given in milliseconds.

Algorithm	Time	Comment
DES encryption	4.9ms/block	4900ms for 1000 encryptions
SHA-1	2.7ms/block	2780ms for a 1000 long hash chain
512 bit RSA key generation	3.4 minutes	
512 bit RSA sig. generation	7028 ms	
512 bit RSA sig. verify	438 ms	$e = 3$
512 bit RSA sig. verify	1376 ms	$e = 65535$
163 bit ECC-DSA key generation	597 ms	
163 bit ECC-DSA sig. generation	776 ms	
163 bit ECC-DSA sig. verify	2448 ms	

Table 1. Timing measurements for cryptographic primitives on the PalmPilot

We first note that generating RSA key pairs is very expensive on the PalmPilot. To generate a 512-bit RSA key pair, three to four minutes of pure computation time is required of the CPU. In addition to inconveniencing the user at wallet setup time, RSA key pair generation drains the PalmPilot's batteries. One may argue that the key pair could be generated on a desktop and downloaded to the PalmPilot, but this limits mobility since setup must take place on the user's PC. If 1024-bit keys are to be generated on the PalmPilot, approximately 20 minutes of pure computation would be required. On the other hand, elliptic curve key pair generation is about two orders of magnitude faster.

As might be expected, DES encryption and SHA-1 hashing are relatively fast as compared with signatures. This suggests that an efficient implementation of a payment protocol for a PDA-based platform should attempt to take advantage of these operations in favor of signatures as much as possible.

RSA signature generation is somewhat slow, taking approximately seven seconds. Our payment protocols take about 900ms seconds when stripped of the cryptography. Hence, RSA signature generation time is significant in comparison to the entire transaction time.

[2] The performance figures for DES, SHA-1, and RSA operations were obtained using Ian Goldberg's port of SSLeay. The figures for ECC operations were obtained using Certicom's Security Builder SDK Release 2.1

Our choice of using 512 bits RSA, rather than 1024, is due to the fact that our application is targeted towards small payments. For small payments, 512 bits may provide adequate security.

The figures for 163-bit ECC-DSA key generation and signature generation are considerably more efficient than the corresponding 512-bit RSA operations. This suggests that ECC-based key generation and signatures are more feasible for use in commerce protocols on the PalmPilot. ECC-based verification does, however, take about 40 percent longer than RSA-based verification.

To summarize, there exists an asymmetry between the performance of RSA and ECC operations. We would like our payment protocols to be as fast as possible, and both signature generation and verifications may be necessary parts of the protocols. If we need to generate signatures on the PalmPilot, we would like to generate ECC-based signatures, since ECC-based signature generation is faster than RSA signature generation. On the other hand, if we need to verify signatures on the PalmPilot, we would like to use RSA since RSA signature verification is faster than ECC-based signature verification. We discuss how our payment protocols take advantage of this asymmetry in Section 3.2.

2.2 Authentication

Common smart cards do not have any means of directly communicating with their owner. They must do so through an untrusted card reader. As a result, it is difficult for the owner to authenticate herself to the card. Often, smartcards require the owner to enter a password. However, there is nothing preventing the machine operating the card reader from recording and replaying the password [10]. The smartcard cannot use challenge-response authentication since humans are incapable of participating in such protocols.

On the other hand, a PDA has *a direct line of communication* with its owner. Before executing a transaction, the wallet controller prompts the user for a password. This password is used for two purposes: first, it authenticates the owner to the PDA. Second, as we shall see in the next section, we use the password to decrypt specific instrument data. Note that the user is prompted for a password only once a transaction is about to take place. Prior to this, the PDA may freely communicate with the vendor.

Unfortunately, entering a password on the PalmPilot is somewhat painful. Clearly, it is desirable to enter the password in no-echo mode. However, since characters are entered using error prone Graffiti, no-echo makes it hard to correctly enter the password. We note that the standard security utility on the PalmPilot prompts the user to enter a password in full echo mode!

2.3 Memory Management and Backups

Since Palm OS provides very limited access control, one must ensure that malicious applications do not have access to private financial information. They must also be prevented from tampering with such data. Our solution is to store all instrument data in encrypted form. This is automatically done by the Instrument Manager [7]. Encryption is done using the user's password which must be entered in order to initiate a transaction. We also append a cryptographic checksum to ensure data integrity.

Nevertheless, malicious applications can delete encrypted instrument data blocks due to the lack of access control in PalmOS. There is no way to prevent this other than to rely on a backup copy stored on a desktop PC. The HotSync Manager application distributed with the PalmPilot automatically backs-up these databases. In the case that the wallet application discovers corrupted data, it may ask the user to perform a HotSync with the desktop. Of course, care must be taken to ensure that already spent digital cash is not doubly spent after the backup process.

3 PDA-PayWord: A Payment Scheme Optimized for a PDA

The lack of tamper resistance on a PDA is unfortunate, although not fatal. Tamper resistance on smartcards can simplify payment protocols such as those used in stored value techniques (see e.g. Mondex [18]). However, one must keep in mind that tamper resistance is not impenetrable (see [3]). As a result, payment schemes on smartcards must also have some mechanisms in place to discourage tampering. Here, a PDA has an advantage over a smartcard: it has more memory and processing power, and can incorporate stronger mechanisms to discourage tampering. We chose to avoid stored value techniques and instead focused on a hash-chain-based digital cash scheme based on PayWord [21]. PDA-PayWord is our implementation of PayWord.

Hash-chain-based micropayment schemes have been studied extensively. The idea of using hash chains to reduce the number of signature generations was independently discovered by a number of groups. These include PayWord [21], Pederson's tick payments [19], micro-iKP [12] and NetCard [2]. Jutla and Yung [14] study a variation where hash trees are used rather than chains. Hash chains are based on a technique due to Lamport [15].

Since we wish to focus on the PalmPilot (user) to vendor interaction (rather than vendor to bank), we allow the bank and vendor to reside on the same machine. This tight coupling between the vendor and bank can be broken at any point in the future.

3.1 Overview of PDA-PayWord

This section contains a high-level overview of PDA-PayWord. A discussion of the design trade-offs involved in PDA-PayWord is given in the next section, and a detailed discussion of our PalmPilot implementation can be found in Section 4.3.

PDA-PayWord is a pre-pay, vendor-specific variant of a hash chain system. To setup a chain of coins the PalmPilot generates $Y_k = h^{(k)}(Y_0)$ where $h^{(i)}$ denotes a repeated iteration of a hash function (we use SHA-1). Y_0 is kept on the PalmPilot as part of a *hash chain instrument*. The PalmPilot sends Y_k, k, d to the bank. The bank returns a *hash chain certificate* containing Y_k, k, d, a vendor-id, an expiration date, and a signature of the certificate using the bank's private key. k is the pre-paid height of the hash chain and d is the denomination of each Y_i. To spend the ith coin when Y_{k-j} was the last coin spent, the PalmPilot sends $Y_{k-j-i} = h^{(k-j-i)}(Y_0)$, i, and the hash chain certificate to the vendor. Initially $j = 0$, and $i = 1$. Recall, these are vendor specific chains and the vendor can verify that the PalmPilot is not double-spending or over-spending.

3.2 Discussion of PDA-PayWord Design Choices

As described in [21], PayWord was designed to amortize the cost of signatures across multiple purchases and minimize on-line communication. In addition to these goals, PDA-PayWord: 1) minimizes the number of and time spent executing cryptographic operations on the user's wallet, and 2) minimizes storage requirements for the hash chain instrument on the user's wallet.

In PDA-Payword, the user's wallet must first generate a hash chain instrument and obtain a hash chain certificate from the bank that binds the "coins" in the hash chain to real value. In this withdrawal phase of the protocol, the user's wallet sends Y_k, k and d to the bank to request a withdrawal of $k * d$ dollars. The bank needs to ensure that the request is being made by the user herself, and therefore PDA-Payword requires that this widthdrawal request be signed. Since this signature needs to take place on the PalmPilot, the withdrawal request is signed using an ECC-DSA signature (as opposed to an RSA signature) to minimize the signature generation time. (Recall that from Table 1, ECC-DSA signature generation takes 776ms on average, while RSA signature generation takes over 7000ms.)

After the bank receives the withdrawal request and verifies the user's signature, the bank then generates a hash chain certificate, signs it, and sends it to the user's wallet. The user's wallet then needs to verify that the bank's signature on the certificate is authentic. This verification takes place on the PalmPilot, and we would like the verification to be as efficient as possible. As such, in PDA-PayWord, the bank signs the hash chain certificate using a RSA signature (as opposed to an ECC signature) since RSA verification is more efficient on the PalmPilot than ECC-DSA signature verification. By taking advantage of the asymmetery between the performance of RSA and ECC signature and verification times, we are able to minimize the amount of time spent on the PalmPilot executing cryptographic operations.

To achieve our second goal of minimizing storage requirements, PDA-PayWord only stores Y_0 as part of the instrument data in the user's wallet, rather than the entire hash chain. The chain is recomputed for every purchase transaction. This is done to save storage space on the PalmPilot, and incurs minimal cost since hashing on the PalmPilot is sufficiently fast. On a smartcard, one might have to store every ith payword along the hash chain since hashing may not be as quick. Of course, a PalmPilot could also take advantage of this technique to reduce the time spent hashing if the user decides to give the wallet more memory in which to run. In our implementation, we conservatively assumed the user would not want the wallet to use more than the minimal amount of storage necessary. Also, thanks to its speed, the PalmPilot can manage longer hash chains than a smartcard.

4 System Design

We briefly summarize some of our design choices. We demonstrate the user interface as well as explain the internal implementation of the various protocols involved. In Section 4.2, we describe our first vendor, a vending machine in our building.

4.1 Wallet Design & Implementation

Our SWAPEROO-based implementation consisted of a digital wallet application written in C++ for the PalmPilot, and a Java implementation of the vendor application. Timing measurements were obtained by running the user's wallet on a PalmPilot III with 2MB of RAM. The PalmPilot III runs on a Motorola DragonBall chip at 16MHz. The bank/vendor application runs on a 300 MHz Pentium PC with 64MB of RAM.

Communication between the PalmPilot and the PC was conducted using TCP/IP over a serial RS232 communications channel.

User Interface The user interface is a collection of forms running on the PalmPilot. When interacting with the vendor, they guide the user through the choice of available items for sale. The user chooses an item and is then asked to choose the payment method, i.e. the payment instrument to be used for the purchase. Once the instrument is selected, the appropriate purchase protocol is activated. As a first step, the user is asked to enter a password to decrypt the instrument data.

A subset of the forms guiding to the user through its interaction with the vendor are presented in Figure 1[3].

Product negotiation We implemented a simple protocol enabling a vendor to download a set of items along with their prices onto the PalmPilot. This protocol is used to convey the customer's choice back to the vendor. Once the required item is agreed upon by both parties the wallet controllers on both the vendor and client take control and execute the transaction.

[3] These figures were produced by taking a snapshot of our wallet application running under the copilot emulator.

Instrument and Protocol Negotiation Instrument negotiation is done by computing the intersection of available instrument types on the vendor and the customer wallets. The set of instruments in the intersection is presented to the user who chooses one of them. We hope the list of available instruments will increase over time. Once the transaction is initiated, the two parties negotiate which protocol is to be used to carry out the transaction (for a given payment instrument there may be multiple protocols offering different levels of efficiency and security). Protocol negotiation is done via a Protocol Negotiation Protocol (PNP) as explained in [7].

Instrument Storage on the PalmPilot Instruments carry some financial information with them. For example, in the case of a PDA-PayWord instrument, the data includes the bottom end of the hash chain, the current position on the hash chain to be spent, and other parameters as well. Each of our instruments is implemented as an encrypted database on the PalmPilot. This data resides in non-volatile memory and is backed up to the desktop during a HotSync.

Link Encryption At the moment the implementation assumes all transactions are local, i.e. the PalmPilot is placed into the vendor's cradle. The PalmPilot does not establish a secure link to the vendor since there is little concern that a direct line between the PalmPilot and vendor can be tapped.

4.2 The Pony Vending Machine

The Pony vending machine[4] supports an interface through a proprietary connection. We were able to interface with it and send it commands through a laptop-PC which serves as a proxy to it. The laptop-PC runs the vendor application, and may issue commands to the Pony vending machine through its proprietary interface. The digital wallet application running on the PalmPilot connects to the vendor application running on the laptop-PC.

When the PalmPilot connects to the Pony-Vendor (the laptop-PC), the user can choose from a list of available products. The set of available products is downloaded onto the PalmPilot as soon as the connection is set up. The user then tells the PalmPilot directly which item to purchase and which instrument to use for payment. The PalmPilot then initiates a purchase protocol with the Pony-Vendor. Once the appropriate funds are transfered to the Pony-Vendor it instructs the Pony-vending machine to dispense the required item. The connection is then closed and the Pony-Vendor is ready to service the next customer.

[4] The Pony (short for Prancing Pony) is a vending machine on the fourth floor of the Gates Computer Science Building on the Stanford campus. Similar to the Prancing Pony in J.R.R. Tolkien's "The Fellowship of the Rings," the Pony vending machine is a common place which those in need of nourishment may go to.

Fig. 1. Example interaction with the vendor. The left form displays available items for sale. The right displays the user's choice and the list of available payment instruments. The textual description of items will eventually be replaced by icons.

4.3 PDA-PayWord Implementation Details

Withdrawal Protocol Before being able to make purchases from the vendor, the user must first make a "withdrawal" from the bank. In our current system, the user takes her PalmPilot to a bank teller and inserts it into a cradle attached to the bank teller's computer. The user authenticates herself by entering her password. The user then enters the withdrawal amount on the PalmPilot, and the PalmPilot generates a *hash chain instrument*. It then executes the withdrawal protocol. Since our implementation uses 5-cent denominations ($d = 5$), the wallet divides the withdrawal amount by 5, yielding the hash chain size, k. The wallet then constructs a hash chain instrument by generating Y_0 ... Y_k. Average hash chain generation times for various dollar amounts using our 5-cent denomination hash chain instrument are shown in Table 2.

For amounts up to $10, a single 5-cent denomination hash chain instrument might provide acceptable performance on the PalmPilot; for larger amounts, using multiple hash-chains with larger denominations might be necessary to achieve acceptable performance.

The user's wallet then kicks off the withdrawal protocol by sending a signed *withdrawal request* to the bank. The user's password is used to decrypt her private key, and the private key is used to sign the withdrawal request. For efficiency,

Amount ($)	Hash Chain Size (words)	Average time (ms)
5	100	504
10	200	896
20	400	1667
50	1000	3970

Table 2. Average Hash Chain Generation Timing Results

an ECC-DSA signature is used to sign the withdrawal request. The withdrawal request is shown in Table 3.

field:	Y_k	k	d
byte-length:	20	2	2

Table 3. Withdrawal Request Message Format

The bank receives the withdrawal request, and verifies the signature on the request. The bank then either deducts the withdrawal amount ($k * d$) from the user's "bank account," or the user gives the bank teller the amount in physical currency. If the user has the appropriate funds in her account, or gives the bank teller the appropriate amount in non-counterfeit currency, the bank teller approves the transaction. Upon approval, a *hash chain certificate* is sent to the user's wallet and stored in the user's hash chain instrument. Note that the user's hash chain instrument has no value without a signed hash chain certificate. The hash chain certificate is also stored on the vendor's wallet in a vendor-side hash chain instrument that can be quickly looked up during execution of the purchase protocol. The vendor-side hash chain instrument also stores the last Y_i spent by the user. (In the case that the bank teller does not approve the transaction, an error message is sent to the user's wallet.)

Generation of the hash chain certificate requires the bank to generate a signature, and an RSA signature is used to sign the hash chain certificate in PDA-Payword (as explained in Section 3.2); note that this task takes place on the bank-PC. The PalmPilot verifies the bank's signature, but does not need to generate a signature itself. The signature generation time for our 300 MHz Pentium bank-PC was 170ms on average. The total time for the PalmPilot to generate and sign the withdrawal request, send it to the bank, and receive a hash chain certificate in response was 1874ms on the average.

A hash chain certificate is shown in Table 4. Many of the fields in the hash chain certificate have already been described, but a brief explanation of all of them can be found in Table 5.

Using 512-bit RSA signatures on a 20-byte message digest of the certificate, the average length of the entire hash chain certificate is 358 bytes. (The length of a 512-bit RSA hash chain certificate can vary because of the variable subject and

field:	d	k	vid	Y_k	sno	$sublen$	$subject$	$elen$	$expdate$	$siglen$	sig
byte-length:	1	2	1	20	4	2	variable	2	variable	2	variable

Table 4. Hash Chain Certificate Format

d	denomination
k	hash chain size
vid	unique vendor identification number
Y_k	top word of hash chain
sno	unique serial number of this certificate
$sublen$	length of variable subject field
$subject$	"subject" of this certificate. currently used to store a comment.
$elen$	length of variable expiration date field
$expdate$	text representation of expiration date in "UNIX" format
$siglen$	length of variable signature field
sig	RSA signature of SHA-1 message digest of this certificate

Table 5. Description of Hash Chain Certificate Fields

expiration date fields.) The bulk of the certificate is made up of the signature field, which is 302 bytes. Although a 512-bit RSA signature on a 20 byte message digest is only 64 bytes, the signature field also contains the bank's public key which can be used to verify the signature (81 bytes), and a public key certificate (151 bytes) signed by the certificate authority that can be used to verify the bank's public key. The CA's public key is hard coded in the wallet application.

Once the hash chain certificate is received by the user's wallet, the wallet verifies the hash chain certificate. Verification of the hash chain certificate took approximately 1008ms in our implementation. Note that two RSA verifications are necessary to check that a hash chain certificate is authentic: one verification using the bank's public key to check the signature on the certificate, and one verification using the CA's public key to check the signature on the bank's public key certificate. The RSA verifications take about 438ms each (for e=3), and hence a total hash chain certificate verification time of 1008ms is reasonable.

If verification of the hash chain certificate is successful, the certificate becomes part of the user's hash chain instrument. (If the hash chain certificate does not verify, an error message is displayed, and the user should demand her money back!)

To summarize, the key components contributing to the total transaction time of the withdrawal protocol are 1) hashing required to construct the instrument, 2) ECC-DSA signature of the withdrawal request, 3) communication overhead between the PalmPilot and the bank, and 4) verification of the hash chain certificate. The time reqiured for (1) is shown in Table 2, and is dependent upon the amount and denomination being withdrawn. The time required for (2) and

(3) combined is approximately 1874ms, and the time required for (4) is approximately 1008ms.

Hence, withdrawal times range from just under 3.5 seconds to just under 7 seconds for instrument amounts from 5$ to 50$. From a usability standpoint, these withdrawal times would seem to be acceptable for most users. [5]

Purchase Protocol Once the hash chain instrument's construction is complete by adding the hash chain certificate to it, a purchase may be accomplished by having the user's wallet send a *purchase request* to the vendor application. If the user has already spent j hash words, then to spend the next i words, the user's wallet sends the purchase request message shown in Table 6 to the vendor application.

field:	Y_{k-j-i}	i	*hash-chain-certificate*
byte-length:	20	2	variable

Table 6. Purchase Request Message Format

The number of hash words to spend, $i = price/d$, is determined based on the price of the product the user wishes to buy and the denomination of the hash chain instrument. Since the hash chain instrument on the user's wallet only stores Y_0, the user's wallet needs to compute $k - j - i$ hashes to construct the purchase request message.

instrument amount($)	*hash iterations required* (words)	*transaction time*(ms)
5	70	1090
10	170	1467
20	370	2267
50	970	4580

Table 7. Purchase Transaction Times

After the vendor wallet receives the purchase request, it retrieves the corresponding vendor hash chain instrument from its Instrument Manager and checks that $h^{(i)}(Y_{k-j-i})$ is equal to the Y_{k-j} that it had stored after the instrument's last use. Note that the first time the instrument is used, $j = 0$ and the vendor

[5] Indeed, for larger amounts, the instrument creation time becomes the dominating factor and can be reduced by using larger denomination coins (choosing a larger value for d). Using multiple hash chains, each with a different d, can also be employed to reduce the amount of hashing necessary, but would necessitate the verification of multiple hash chain certificates.

wallet checks that $h^{(i)}(Y_{k-i}) = Y_k$. The vendor wallet also checks the validity of the hash chain certificate that is presented in the purchase request, and if the certificate is valid, the vendor wallet stores Y_{k-j-i} in its vendor-side hash chain instrument and sends back "OK" as the *purchase response*. If any of the above checks fail, an "ERROR" purchase response is sent back to the user wallet.

Purchase protocol timing measurements to do a first-time $1.50 buy with 5-cent denomination hash chain instruments of varying initial sizes are presented in Table 7.

5 Summary and Conclusions

Our experiments show that a PDA may be viewed as a *portable commerce device without tamper resistance* that is suitable for small payments. PDAs are computationally more powerful than smartcards. However, since they do not contain cryptographic accelerators certain operations take longer, and the performance of cryptographic primitives need to be taken into account when designing commerce protocols for PDA's. For instance, RSA signature generation is slow, while RSA verification is fast. On the other hand, ECC-based signature generation is fast while verification is slow.

In conclusion, commerce protocols can be implemented to perform transactions efficiently on a PDA platform. PDA-PayWord is an example of an implementation of a commerce protocol that is designed to achieve acceptable performance on a PDA by taking advantage of the performance characteristics of both RSA and elliptic curve cryptograpy on the PalmPilot.

Acknowledgments

Special thanks to Andrew Toy for revitalizing the PC interface to the Pony vending machine.

References

1. 3Com Corporation, `http://www.3com.com`
2. R. Anderson, C. Manifavas, C. Sutherland, "Netcard – a practical electronic cash system", in Fourth Cambridge Workshop on Security Protocols.
3. R. Anderson, M. Kuhn, "Tamper resistance – a cautionary note", In 2nd USENIX Workshop on electronic commerce, 1996, pp. 1–11.
4. Certicom Co., `http://www.certicom.com`
5. B. Cox, D. Tygar, M. Sirbu, "NetBill security and transaction protocol", `http://www.ini.cmu.edu/netbill/home.html`.
6. CyberCash, `http://www.cybercash.com`
7. N. Daswani, D. Boneh, H. Garcia-Molina, S. Ketchpel, A. Paepcke "SWAPEROO: A Simple Wallet Architecture for Payments, Exchanges, Refunds, and Other Operations", Proceedings of the Third USENIX Workshop on Electronic Commerce, 1998.

8. DigiCash, http://www.digicash.com
9. Financial applications for the PalmPilot, http://www.pilotzone.com/prod_business.html
10. A. Glass, "Could the smartcard be dumb", Proceedings of Eurocrypt '86, 1986.
11. I. Goldberg, pilotSSLeay-2.01, http://www.isaac.cs.berkeley.edu/pilot/
12. R. Hauser, M. Steiner, M. Waidner, "Micro-payments based on iKP", In 14th Wolrd-wide congress on computer and communication security protection, 1996.
13. S. Jarecki, A. Odlyzko, "An efficient micropayment system based on probabilistic polling", Proc. Financial Cryptography '97.
14. C. Jutla, M. Yung, "Paytree: amortized signature for flexible micropayments", In 2nd USENIX Workshop on electronic commerce, 1996, pp. 213–221.
15. L. Lamport, "Password authentication with insecure communication", Communications of the ACM, vol. 24 (11), 1981, pp. 770-771.
16. R. Lipton, R. Ostrovsky, "Micro-Payments via efficient coin flipping", Proc. Financial Cryptography '98.
17. M. Manasse, "The Millicent protocols for electronic commerce", Proc. of the 1st USENIX workshop on Electronic Commerce.
18. Mondex International, http://www.mondex.com
19. T. Pederson, "Electronic payments of small amounts", Tech. Report, DAIMI PB-495, Aarhus University, Computer Science Dept., Aug. 1995.
20. R. Rivest, "Lottery tickets as Micro-Cash", Financial Cryptography '97.
21. R. Rivest, A. Shamir, "PayWord and MicroMint", CryptoBytes, Vol. 2, No. 1, pp. 7–11. Available from http://www.rsa.com
22. Security applications for the PalmPilot, http://www.pilotzone.com/util_security.html
23. Secure Electronic Transactions. http://www.setco.org
24. Y. Yacobi, "On the continuum between on-line and off-line e-cash systems – I", Proc. Financial Cryptography '97.
25. Windows CE 2.0, http://www.microsoft.com/windowsce/