Open APIs for Embedded Security

Carl A. Gunter

Department of Computer and Information Science
University of Pennsylvania
3330 Walnut Street
Philadelphia PA 19104-6389
gunter@cis.upenn.edu
http://www.cis.upenn.edu/gunter

Abstract. Embedded computer control is increasingly common in appliances, vehicles, communication devices, medical instruments, and many other systems. Some embedded computer systems enable users to obtain their own programs from parties other than the maker of the device. For instance, PDAs and some cell phones offer an open application programming interface that enables users to better customize devices to their needs and support an industry of independent software vendors. This kind of flexibility will be more difficult for other kinds of embedded devices where safety and security are a greater risk. This paper discusses some of the challenges and architectural options for open APIs for embedded systems. These issues are illustrated through an approach to implementing secure programmable payment cards.

1 Introduction

Embedded computer systems are computers that are installed in devices such as appliances, vehicles, cell phones, medical devices, and so on. They typically differ from computers in servers and desktop systems like PCs because of limits on size, power consumption, form factor, and location (*eg.* mobility), resulting in limits on computational power, memory, and communication connectivity. Embedded systems are increasingly common; they control important devices in military, government, industrial, and, increasingly, consumer contexts. Because of the many constraints on such devices and the fact that they are used in contexts with safety and security concerns, embedded systems typically do not enjoy many of the desirable features of servers and desktop systems. For instance, a computer embedded in a car or vacuum cleaner probably is not accessible from the Internet, and its software probably cannot be easily updated to new versions. On the other hand, computers in PDAs are somewhat like desktop systems except they trade off power, connectivity, and features like a large monitor to enhance mobility. In particular, current PDAs have in common with desktop computer systems the ability to run programs developed by parties other than the maker of the PDA. Such programs can be installed by the user of the device after she has purchased it. Most embedded computers, such as those in appliances

and vehicles, do not offer this level of user control and flexibility. Some systems are at a boundary in this respect. For instance, there are now cell phones that allow users to download and run programs from third parties. Common software engineering practice leads software developers to create software in layers and such layered systems often provide an *Application Programming Interface (API)* to aid software evolution. The APIs may also enable the device vendor to work more easily with subcontractors to obtain application software for their platform. The key difference that is the interest of this paper is what it takes to allow the API to be *open* so that application developers not under the direct control of the platform vendor can provide programs to customize the platform.

There are a collection of barriers that prevent the deployment of open APIs. Many of these barriers are commercial: platform vendors often consider it more profitable to write their own applications, or may be concerned about losing control of their platform if its API is open. There are also many technical challenges. This paper focuses on ways in which control can be balanced between the embedded computer, its host device, and remote host devices. The discussion is divided into seven sections. Section 2 considers various options for delivering code to an embedded device with an open API. Section 3 focuses on the tradeoffs in using remote control and illustrates this with smart cards implementing the SET payment protocol. Section 4 introduces the concept of a programmable payment card. Section 5 surveys technologies that could aid the implementation of such cards. Section 6 introduces the refinement architecture and the filter implementation as a means of realizing programmable payment cards. Section 7 concludes.

2 Delivery Architectures

One of the challenges for open APIs is how to *deliver* code to the device. Because of the diverse nature of the contexts in which embedded systems are used, there is a similar diversity of challenges and options for delivery. Some of the basic options are shown in Figure 1. The primary components of the system are the *embedded computer* itself and its *user*. The embedded computer is contained within a *host device*, which could be a car, a vacuum cleaner, a cell phone, or many similar devices. The embedded computer may be permanently encased in the host device, as in most appliances, or it may be removable, as in smart cards for financial transactions or cell phones. The host may include a capable computer itself, or it may derive its intelligence from the embedded device. If the embedded computer has an open API and the user is able to program it, then it must have some way to access new programs. There are at least four common options. The user may be able to customize the device in rudimentary manner though some input interface provided by the host. For serious programming, the device could accept some kind of *removable media* that carries programming or get its programming across a *network link*. In the networked case, the code could be derived from *remote data* and moved to the device or the program could reside elsewhere and operate by *remote control*.

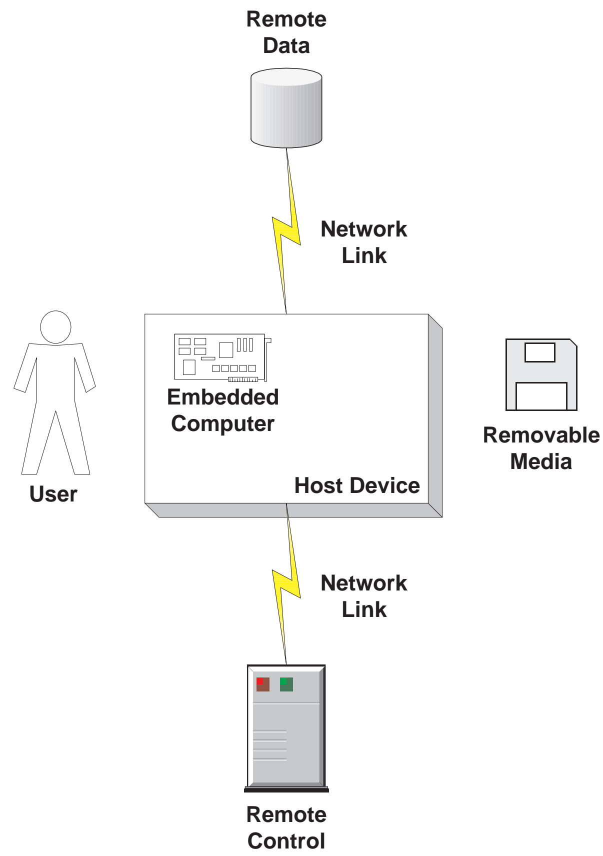

Fig. 1. Communication Options

Typical desktop computers are characterized in part by their easy access to most or all of these options. By contrast, embedded computers often offer a far more limited range of possibilities. We illustrate the options by considering the problem of *programmable cooking devices*, specifically multi-modal ovens. A multi-modal oven is one that offers several different cooking options. For instance, the GE Advantium provides three actuators (a microwave function and two sets of halogen lamps) and two sensors (for heat and humidity). Such ovens offer the ability to perform actions that cannot be performed by current single-mode ovens; for instance, they can cook food quickly like a microwave and brown it like a traditional oven. Other functions like jet impingement can provide faster heat transfer and crust development with more retention of moisture. However, hardware vendors are constrained by the existing means of programming ovens. A typical microwave is programmed by the user keying in a recipe (simple program) taken from a cookbook or off a frozen food package. Such recipes must be very simple given the patience and skill level of the user. A recipe that uses multiple sensors and actuators in a non-trivial way is difficult to describe with this constraint. This is done to some extent. The Advantium can be programmed with three numbers representing cook times for its actuators, and these recipes appear on some kinds of frozen foods. Let us consider some of the alternatives.

A simple approach would be to provide programs in storage on the host device using something like flash memory. The host device would provide a library of recipes from which the user could select. High end microwaves provide this functionality now, at least for a collection of typical recipes. A more ambitious goal would be to provide a library of recipes for frozen foods available off the shelf of the local grocery. This would allow frozen food vendors to use recipes that cook their food better than recipes keyed in by the user from the packages. First of all, it would allow much more complex recipes, possibly taking real advantage of all those fancy sensors and actuators. But even at a more elementary level, the program could help standardize the processing. For instance, recipes often have instructions like ‘cook for 3 to 5 minutes’ because ovens vary from 600 watts to 1000 watts, indicate that the food should be turned in ovens that do not turntables, and so on. To make this process easier, vendors have explored the idea of putting bar code readers on experimental microwave ovens. Linear bar codes are used by stores to identify products based on the Universal Product Code (UPC) standard. The readers for such codes are cheap, so a user could scan a frozen food package using a bar code reader on the oven, and the oven could run the indexed program from its database of recipes. The real drawback to this approach is the fact that changes occur and a static database of recipes will eventually become out-of-date.

Some means is needed to communicate new recipes to the embedded control in the oven. One approach (see US patents 5,812,393 and 5,883,801) is to let the linear barcodes serve as programs themselves. One cannot provide a real programming language with programs that are expressed as 10 digits, but the standardization problem can be partially addressed in this way. For instance, ovens with different power capabilities can translate the supplied parameters

themselves so the user does not need to calculate what the cook time should be for their specific oven. A technique explored by the OpEm Project at Penn is the use of two-dimensional barcodes [7, 9]. These can be printed inexpensively on paper and contain far more data than linear barcodes. A recipe can be represented as a small program of about 1 to 2 kilobytes, placed on a package, and read by the oven using a charged coupled device. This provides programs through inexpensive 'removable media'. The OpEm work explored the use of a subset of Java to control a 'microwave oven object'. It is possible to create interesting recipe bytecodes with one or two kilobytes of space; Java bytecodes are redundant enough to benefit from compression, even for programs of only one kilobyte. There are a variety of benefits in doing this. Aside from addressing the problem with diverse oven capabilities, there are things that a program can do that are too complicated to assign to users, such as adjusting cooking times based on user-instigated pauses.

A commercial disadvantage to 2D barcode programs is the fact that food vendors do not provide such recipes currently, so a 'chicken and egg' problem exists: without barcode recipes on packages there is no benefit to manufacturing or owning an oven that reads barcodes, and, when few ovens read barcodes, there is limited incentive to supply barcode recipes on packages. This problem did not exist with the database solution since one could develop a recipe suite that covered the products from most frozen food vendors and put this on the programmable oven at the outset. If the problem of keeping this database up-to-date can be addressed, then this could provide a practical approach. Microwave vendors have considered adding serial ports to ovens. This would enable the device to download recipes in a manner similar to TiVo (`www.tivo.com`), which gets schedules for television programs by short periodic telephone calls. This essentially corresponds to the remote data approach in Figure 1. A more sophisticated variation would be to put the oven on the Internet. Ovens that attach to computers and can therefore download programs over the Internet were explored in the Rutgers-Samsung IMWO project, and Sharp introduced such an oven in March of 2002 (`http://www.sharp-world.com/corporate/news/020130.html`). It seems plausible that ovens and other kitchen appliances will offer WiFi links in the future.

3 Remote Control

The previous section used programmable microwaves to illustrate each of the delivery approaches in Figure 1 except remote control. In a sense remote control is the anti-thesis of embedded control, since it shifts intelligence from the embedded computer and its host to a remote host. The effects of this shift can be particularly appreciated in the contrast between *magnetic stripe* tokens versus *smart cards*. Both of these are familiar contents of wallets, used for purposes like payment cards and loyalty programs. Magnetic stripe cards contain data and their processing is based solely on control from a local or remote host; smart

cards contain a processor and are able to supply a limited amount of embedded control when inserted in a suitable host port.

An example helps illustrate the distinction. In my wallet I carry a Starbucks card, used to purchase drinks from participating branches of this coffee shop vendor. This is a magnetic stripe card that indexes my account with Starbucks. I periodically put money into the account by giving the card and some cash to a clerk. The card works at branches other than the one that took the cash and acts as a kind of electronic wallet. When I want to make a purchase with the card, the index is used to determine my balance, from which the charges are deducted. By contrast, I once had a smart card embedded into a Penn ID card. This smart card served as an electronic wallet, able to hold a digital representation of money for use with Penn services, including, for instance, the vending machines in my building. The main difference between the magnetic stripe card and the smart card is where the processing is carried out. The magnetic stripe card has no embedded control; it provides only data, possibly encrypted to protect sensitive data like a Personal ID Number (PIN). The smart card, by contrast, is able to do some on-board processing and is able to protect cleartext data on the card by physical means and interface limits as well as encryption. The smart card is especially well-suited to offline operation since it cannot easily be duplicated,[1] so this is a good fit for vending machines, which may not have the option of checking an account balance over a network link.

Network connectivity is a crucial factor in whether remote control is feasible. When it exists, it can contribute significantly to security and simplicity. Network connectivity may partially explain the wider use of smart cards in Europe compared to the United States. Balances for payment cards in the U.S. were initially checked by a telephone call, whereas European transactions commonly relied on the embedded control in a smart card. Let us now consider models for payment card transactions in more detail as a way to study tradeoffs between embedded and remote control. For more on the tradeoffs between magnetic stripes and smart cards see [22].

Payment cards are a major means for carrying out consumer purchases, competing with other means such as cash and checks. They come in several flavors: credit cards provide a loan capability, demand cards allow for payments to be delayed for a month or so, and debit cards deduct costs immediately from a cardhold's bank account. They typically provide for three to five participants, illustrated in Figure 2. A typical scenario is a *cardholder* visiting the premises of a *merchant* such as a department store. The user has obtained his card from an *issuer*. Issuers are often banks such as Citibank or MBNA. The card is inserted into a host (terminal) provided by the merchant, which contacts an entity known as the *acquirer*. The acquirer may be a computer operated by the bank of the merchant. The acquirer contacts a *payment system* with information like the amount of payment requested. Common payment systems include MasterCard

[1] Magnetic stripes are also more vulnerable to offline attack: if data is encrypted with a PIN on a magnetic stripe, it would be easy to check all possible PINs to determine the encrypted value.

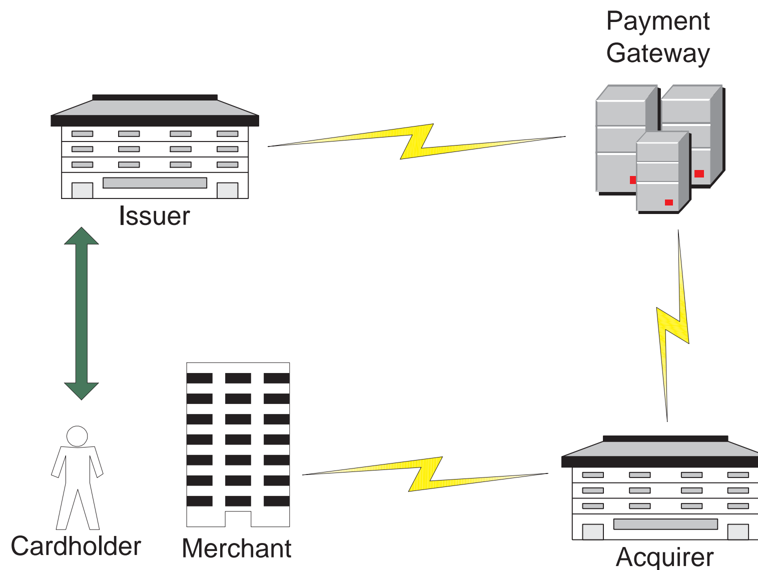

Fig. 2. Common Payment Card Architecture

and Visa. The payment system may contact the issuer of the card holder to determine whether the cardholder has enough money to make the payment. An approval propagates back through the acquirer to the merchant, who obtains an authorizing signature or PIN from the customer and provides him a receipt. There are a number of variations on this scenario. For instance, the issuer may be a store, like Sears, rather than a bank, and the payment system may be different. For example, when American Express is used as the payment system, the merchant contacts the payment system directly rather than contacting an acquirer bank. For our purposes, the main point is the fact that the transaction is distributed, online, realtime, and the card does not provide embedded control to the host of the merchant. For more information about payment cards see [6].

The basic scenario described in Figure 2 has changed in important ways with changes in buying habits. For instance, cardholder users may not be physically present at the site of the merchant. At first this occurred primarily because of telephone purchases, but more recently consumers have used the Internet to make remote purchases. This has a significant impact not only on the convenience of the transaction for the consumer but also on the risks for the merchant and payment system. In particular, fraud in Internet payment card purchases is a major cost. In response to this, a collection of organizations developed a standard for Secure Electronic Commerce (SET) to provide improved protections. Figure 3 illustrates the approach. A *user* runs SET from a *host* computer, which may be her personal computer at home. SET can run from the host computer alone, but it may also be used with embedded control provided by a smart card for running

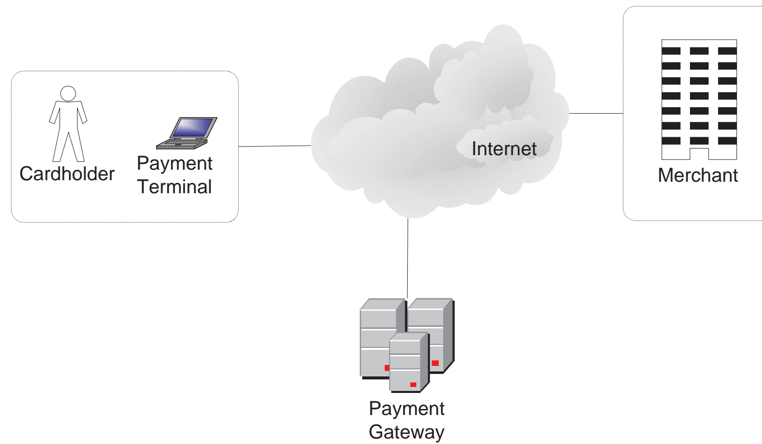

Fig. 3. Internet Payment Card Architecture

(portions of) the SET protocol. The user host and embedded system exchange a series of messages over the Internet with the *merchant*. The merchant, in turn, exchanges SET protocol messages with a *payment gateway* to ensure payment for the transaction.

SET is described in a hefty series of documents [16–19]. We focus on the protocol in which a user C makes a purchase from a merchant M, who is paid by a payment gateway P. Before the protocol begins, the user C and the merchant M negotiate a description OrderDesc of the item to be purchased and an amount PurchAmt to be paid for it. The aim of the protocol is to assure the merchant that he will be paid for the order without revealing what it is to the payment gateway. At the same time, the user and merchant need to convince the payment gateway that it is a legitimate purchase through the use of secret information, namely CardSecret and PANData. The PANData includes information such as the user's Primary Account Number (PAN) that should not be revealed to the merchant. The cryptographic technique is called a *dual signature*. It allows two parties (the merchant and the payment gateway in this case) to participate in (what they can prove to be) the same transaction without revealing all information to both parties. We describe a simplified version of the purchase request protocol based on the formal treatment in [3], which, in turn, was derived from [18]. SET uses PCKS envelops and X.509 certificates. To keep things simple, we omit information in messages about certificates and use a simplified notation. We write $[X]_S$ for message X together with a signature by subject S, and $\{X\}_S$ for message X encrypted for subject S, and $H(X)$ for the hash of message X.

The messages in the SET purchase request are given in Figure 4. In PInitReq, the user sends a challenge Chall_C to the merchant, together with a local identifier LID_M. In PInitRes, the merchant responds to this with a unique transaction identifier XID, a challenge Chall_M of his own, as well as the local identifier and

PInitReq	$C \rightarrow M$:	$\mathrm{LID}_M,\ \mathrm{Chall}_C$
PInitRes	$M \rightarrow C$:	$[\mathrm{LID}_M,\ \mathrm{XID},\ \mathrm{Chall}_C,\ \mathrm{Chall}_M]_M$
PReq	$C \rightarrow M$:	OIDualSign, PIDualSign
AuthReq	$M \rightarrow P$:	$\{[\mathrm{LID}_M,\ \mathrm{XID},\ H(\mathrm{OIData}),\ \mathrm{HOD},\ \mathrm{PIDualSign}]_M\}_P$
AuthRes	$P \rightarrow M$:	$\{[\mathrm{LID}_M,\ \mathrm{XID},\ \mathrm{PurchAmt},\ \mathrm{authCode}]_P\}_M$
PRes	$M \rightarrow C$:	$[\mathrm{LID}_M,\ \mathrm{XID},\ \mathrm{Chall}_C,\ H(\mathrm{PurchAmt})]_M$

where

$$\begin{aligned}
\mathrm{HOD} &= H(\mathrm{OrderDesc},\ \mathrm{PurchAmt})\\
\mathrm{OIData} &= \mathrm{XID},\ \mathrm{Chall}_C,\ \mathrm{HOD},\ \mathrm{Chall}_M\\[1ex]
\mathrm{PIHead} &= \mathrm{LID}_M,\ \mathrm{XID},\ \mathrm{HOD},\ \mathrm{PurchAmt},\ M,\ H(\mathrm{XID},\ \mathrm{CardSecret})\\
\mathrm{PIData} &= \mathrm{PIHead},\ \mathrm{PANData}\\[1ex]
\mathrm{OIDualSign} &= \mathrm{OIData},\ H(\mathrm{PIData})\\
\mathrm{PIDualSign} &= [H(\mathrm{PIData}),\ H(\mathrm{OIData})]_C,\ \{\mathrm{PIHead},\ \mathrm{PANData},\ H(\mathrm{OIData})\}_P
\end{aligned}$$

Fig. 4. SET Purchase Request Protocol

challenge from the user, all signed with his private key. The third message in the protocol, PReq, is its essence. The user sends the merchant the dual signature. It consists of two parts. A first part, OIDualSign, contains information for the merchant. In particular, it includes the order instruction data, OIData, which connects the request to the agreed purchase item and payment amount. A second part, PIDualSign, contains information for the payment gateway. In particular, it connects the request to the payment instruction data PIData, which includes the PANData and a hash of the card secret. Both of these parts are sent to the merchant, who is able to check a signature in PIDualSign to connect the purchase and order information, but is unable to learn the purchase information because it is encrypted under the public key of the payment gateway. The merchant forwards PIDualSign to the payment gateway, who then confirms the transaction.

The SET protocol is characterized by a number of secrets and authentication operations. While the protocol could be implemented on a host PC and these secrets could be stored there, it is beneficial to keep some of the sensitive information on a tamper-resistant smart card. There have been a variety of attempts to support SET with embedded control provided by smart cards. The *Chip Electronic Commerce (CEC)* specification aimed to provide SET for Europay Mastercard Visa (EMV) smart cards. The *Chip SET (C-SET)* pilot from CyberCard provided a translater that enabled SET to work with a smart card. The vWALLET smart card was a SET card developed as part of the e-COMM pilot initiated by Gemplus, VISA International, France Telecom, BNP, Societe Generale and Credit Lyonnais. Recent work has produced a SET implementation for the Java Card [15, 12, 13]. This work, and a number of related technologies for the Java Card, will be discussed further below.

4 Programmable Payment Cards

I now introduce an architecture and application that provides a range of interesting challenges for open embedded control. The application is a *Programmable Payment Card (PPC).* It is common for an issuer, such as a bank, to provide payment cards to an enterprise so the enterprise can enable its employees to make purchases with the cards. This improves efficiency by empowering employees. It does have the limitation that employees must be trusted to some extent to use the cards properly in accordance with policies under which they were given the card. A simple policy would be to restrict the value of the purchases that can be made with a given card within a given time frame such as a month. Another possibility is to issue a card that is specific to a particular vendor, so an employee with a card can purchase only from the authorized vendor. Families also act like enterprises in this respect, offering cards to older children to be used in emergencies or for other purposes. Some policy can be enforced. For instance, a parent could open a bank account and obtain a debit card against this account to prevent a child from spending more than the balance of the account. In general, however, the policies that govern cards in enterprises and families are too complex to be enforced by the issuing bank and the payment gateway. Moreover, issuers and payment gateways may not be eager to know or enforce policies of *secondary issuers* such as enterprises and parents since this adds transaction complexity and may create liabilities. Another factor is privacy: secondary issuers may not want to communicate policies to banks, stores, payment gateways, and other external entities.

Is it possible to design a payment card that could accept policies from secondary issuers after the card was originally issued? For instance, an enterprise could 'create' a card that could be given a specific budget each month, or a family could 'create' a card that could only be used to purchase hotel accommodations. To some extent issuers do try to distinguish their cards by offering special ways to customize them. For instance, there are cards targeted at small businesses that help the owner control risks for employees charged with card-based procurement. If, however, the owner of a small business wanted a policy that was not wanted by thousands of other similar businesses, then it is unlikely that an issuer would be willing to create such a custom card. Looking at Figures 1 and 2, one possibility is to provide for some form of customizable remote control. For instance, we could augment Figure 2 so that the payment gateway not only consulted the '*primary* issuer' (typically a bank) but also the secondary issuer. Suppose, for instance, that Penn wanted to issue cards to employees that could only be used to make purchases from a specified collection of merchants. When an employee attempted a purchase, the payment gateway would contact a server at Penn with information about the purchase. Penn's server would need to approve the request before the payment gateway would authorize the purchase. This approach, based on remote control, has a number of the desired benefits, including significant customizability. It complicates the approval process, however, and the merchant and payment gateway may consider it too cumbersome to include additional online checks involving a diverse range of parties.

Another idea is to use embedded control for policy. In this approach, policy is installed on the card itself by the secondary issuer. This has the significant limit that it only works for payment cards that are smart cards using a protocol like SET, with a sufficient amount of memory and protection to enforce the policy. The rest of this paper will argue that this is at least *technically* feasible at the current time in a limited way, and trends will make it somewhat more feasible in the future. It provides a non-trivial case study in the analysis of embedded control architectures and assurance technologies.

5 Open Smart Cards

The value of an open API for smart cards was recognized in the 1990's and, in particular, the Java Card is a widely accepted instantiation of this today. This section sketches background for some of the relevant technologies that could contribute to the development of PPCs. These technologies include: smart cards, the Java Card, the GlobalPlatform, byte code verification on the Java Card, and payment protocols on the Java Card.

Smart Cards

Smart cards, also known as integrated circuit cards, where invented in the late 1960's and are now commonly used for personal identification, payment, communication, and physical access applications. There are many smart card vendors and the ISO 7816 series of standards provides an industry-wide baseline. There are a number of kinds of smart cards. Our focus is on microprocessor contact cards. These include a microprocessor that enables the card to do certain kinds of calculations and a set of contacts that allow the card to get power and communicate through a Card Acceptance Device (CAD). They are commonly distributed in a credit-card-sized plastic substrate and provide a degree of tamper-resistance against physical efforts to learn the contents of the card memory or subvert its computational functions. Aside from the ISO 7816 standards there are standards for smart card Subscriber Identity Modules (SIMs) in telephones that support the Global System for Mobile Communications (GSM), as defined by the European Telecommunications Standards Institute (ETSI). These include GSM 11.11 and GSM 11.14, which define interfaces and toolkits respectively, and GSM 03.19, which defines a SIM API for the Java Card platform. Also, EMV provides standards for cards to suit the needs of the financial industry (`www.emvco.com`).

Smart cards typically provide three kinds of memory: Read Only Memory (ROM), Electrical Erasable Programmable Read-Only Memory (EEPROM), and RAM (Random Access Memory). ROM is used to store fixed programming and parameters. It holds data even without power, but cannot be written after the card is fabricated. ROM holds the OS of the card and permanent applications. A typical card might have 64 kilobytes of ROM. EEPROM can also be preserved when the card has no power and, unlike ROM, it can be modified

during the service life of the card. It can be used to hold data and applications that are added after the card is made. However, it is slow to write to EEPROM and it supports only a limited number of such writes, so EEPROM is not appropriate for often-changing variables. A typical card might have 16 kilobytes of EEPROM. RAM provides the necessary workspace for computation. It is quick to modify RAM, and it does not wear out with many modifications. However, RAM is expensive on a smart card, and memory in RAM is lost when the card is not powered. A typical card may have only 1 kilobyte of RAM.

Java Cards

An API for using Java to program smart cards was introduced in 1996 by Slumberger. This effort was expanded to include other companies in an industry consortium called the Java Card Forum. This has evolved to a collection of specifications supported by Sun (`java.sun.com/products/javacard`) covering the Java Card API, the Java Card Runtime Environment (JCRE), and the Java Card Virtual Machine (JCVM). The current specification is 2.2 and it offers a good platform for open embedded programming. In particular, it supports a restricted subset of Java that can be compiled to JCVM byte code and a run-time system that enables this code to run in a restricted *context*. The rules for communicating between contexts enables multi-application programming with a limited degree of sharing.

The Java Card supports many of the familiar Java programming constructs such as packages, classes, interfaces, exceptions, inheritance, and dynamic object creation. It has a limited collection of data structures including booleans, bytes, short integers and one-dimensional arrays, but not long integers, double, float, characters, strings, or multi-dimensional arrays. There is no dynamic class loading, object serialization, or threads and no garbage collection. The Java Card does not support the Java Security Manager, but instead provides *applet firewalls,* which, as mentioned above, protect distinct groups of applets through contexts.

Other approaches to open multi-application smart cards include MultOS (`www.multos.com`) and Smart Card for Windows.

GlobalPlatform

Java Card programs are created by first making Java bytecode and then processing this to create a Converted APplet (CAP), which is then loaded and installed on the card. An especially important aspect of this process is the fact that type checking is carried out in these steps rather than on the card. This raises a challenge if the development environment is not trusted by the card. In a multi-application open card this is not an unusual state of affairs since applications from different vendors may not trust each other. One approach to deal with this problem is to rely on the issuer to certify that programs are well-formed and secure.

Visa developed an architecture in the 1990s to instantiate this approach. The architecture was first called the Open Platform and is now subsumed in an industry consortium called the GlobalPlatform (`www.globalplatform.org`). The architecture is intended to be independent of the underlying smart card runtime system, but assumes that the system supports features like the ability to protect confidentiality and integrity between applications installed post-issuance by parties that do not wish to trust each other. It also aims to keep the issuer in substantial control of the card while not requiring the issuer to know all details of the providers. Some examples help motivate the aims. Suppose a merchant would like to provide a loyalty application that gives the user credit for shopping with that merchant. If a patron is visiting the merchant, it would be convenient to use the terminal of the merchant to download the new application to the patron's card without the need to contact the card issuer. Referring to Figure 1, this enables software to be provided from the local host without the need to contact some kind of remote storage, or at least to contact remote storage under the control of the party that owns the local host. Another class of examples that motivate the GlobalPlaform are applications that hold sensitive information. Examples include a car rental company that keeps information about a driver or a medical application that keeps health care and insurance records.

This GlobalPlatform architecture is illustrated in Figure 5. A *Runtime En-*

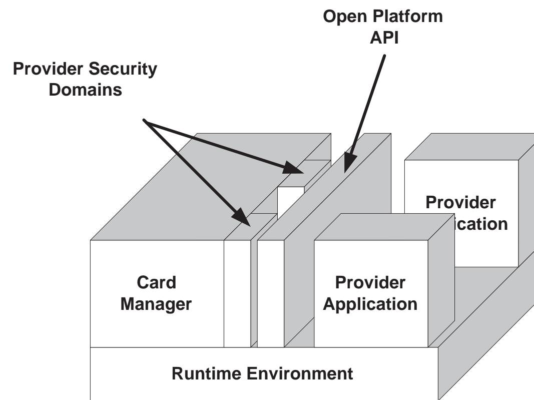

Fig. 5. GlobalPlatform Architecture

vironment underlies the system, and card functions are controlled by a *Card Manager*. The Card Manager uses a collection of *Provider Security Domains* to determine the rights of parties to add *Provider Applications* to the card using the *Open Platform API*. Each security domain includes keys needed to authenticate the provider through a CAD before authorizing the installation of the provider's

application. This authentication is carried out by the card manager through the functions in the API.

The problem of how to ensure that programs satisfy the necessary properties before they are installed and how to place control in the hands of the issuer is illustrated in the steps involved in how the provider loads and installs an application. This process is illustrated in Figure 6. In Step 1, the issuer produces

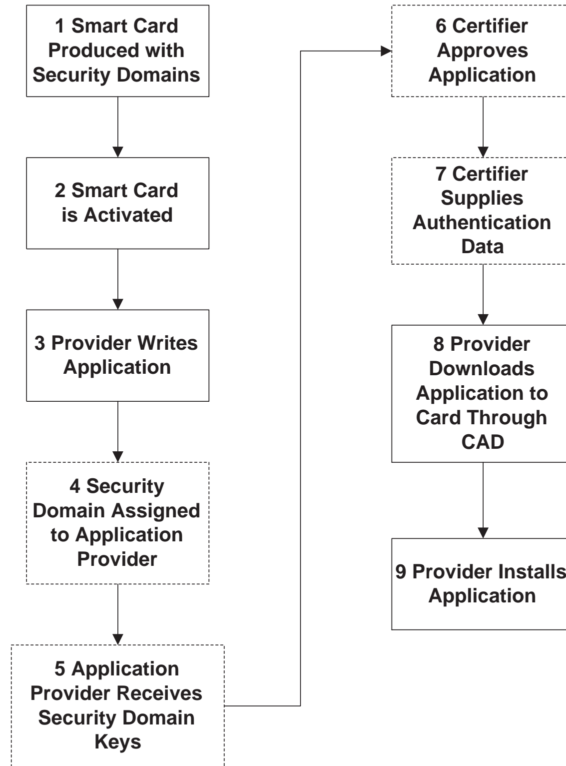

Fig. 6. Steps in the GlobalPlatform Provider Loading

a card with a collection of security domains and, in Step 2, the card is activated. In Step 3 the provider creates an application and, in Step 4, obtains the rights to a security domain. In Step 5, the provider obtains the necessary cryptographic keys to prove her rights. The application provider submits her program to a certifier in Step 6, who reviews it for security and other concerns. The submitted program does not need to contain private information of the party that will receive the card. Moreover, the certifier could be the issuer or an independent certification authority. In Step 7, the certifier then supplies necessary authentication data such as a signature on the program that can be checked by the card manager. In

Step 8, the provider uses a CAD to download the approved application onto the card, which is installed in Step 9 once all of the security checks have succeeded.

5.1 Byte Code Verification on Java Cards

The use of certification by digital signature to ensure security features of provider applications has a number of drawbacks. In particular, it takes back some of the primary benefit of post-issuance installation of applications by making it contingent on Steps 4-7 indicated with dotted lines in Figure 6. Eliminating these steps by developing some form of practical on-card verification has been an interesting research objective for the last 5 years. One possibility is to use a *defensive* virtual machine on the card that checks types at runtime. This is expensive, however. If the types are checked statically off-card, then there is no way for the card to ensure security without repeating the checks, so on-card static verification is the only alternative. Static verification of Java programs in the JVM implementation is too expensive to perform on-card, so some alternative is needed. One approach is to augment the usual verification so that it provides 'hints' to aid re-verification on the card. This approach [20, 8] can be realized by providing type maps with additional type information for stack and register contents as a supplement to the usual Java byte code. Although generating the type maps may be expensive, it is easier to verify the code with them, so this can be exploited by the on-card verification. A practical way to structure this is shown in Figure 7. The basic idea is to insert an 'assurance layer' at the end of the off-card code generation and before the on-card installation. The figure depicts the following steps. In Step 1 a Java compiler creates a class file, which is input in Step 2 to a CAP converter to create a standard CAP file. In Step 3 this CAP file is converted to include assurance information such as a type map that can be processed in Step 4 to confirm on-card that the code is well-typed. This step uses the type map information to deal with the limitations of on-card verification. The verified CAP file is now used in Step 5 to create a verified applet that can be run in a non-defensive virtual machine. A similar strategy was developed for the Sun virtual machine for mobile devices [23], where a 'preverifier' is used to create the type maps.

Another strategy for on-card verification based on the steps in Figure 7 is described in [11]. The idea in this case is to transform the CAP file into a form where byte code verification can be carried out on-card. The basic observation is that two assumptions about the Java program to be processed and one assumption about the Java runtime environment suffice to eliminate the space overhead imposed by type maps. The runtime environment is assumed to initialize non-parameter registers on method entry to a safe value such as a null reference. Assuming that this is satisfied, the output of a standard Java compiler can be transformed to satisfy the two assumptions on programs: the operand stack is empty at all branch and branch target instructions, and, for each method evaluation, each register has only one type. If one accepts that the restriction on the runtime is acceptable for JCRE, then the main questions concern: (1) the overhead imposed by the transformations to achieve the other two assumptions,

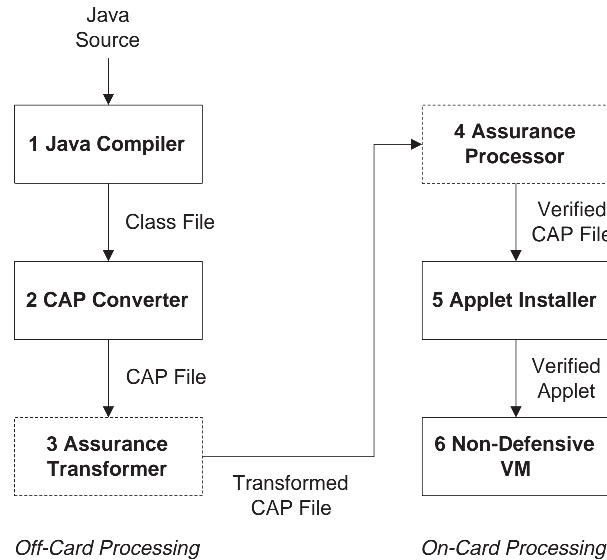

Fig. 7. Steps in On-Card Byte Code Verification for Smart Cards

and (2) the cost of the on-card verifier in time and memory. Tests show [11] that the transformation increases code sizes only slightly (less than 3%) if at all. The space required by the verifier code can be estimated at 10 kilobytes, and it is estimated that verification can be carried out on a Java Card at one kilobyte per second. This can be compared to an EEPROM budget of 32-64 kilobytes for all provider applications.

Payment Protocols for the Java Card

With the GlobalPlatform architecture and technologies for on-card verification there is a range of possibilities for ensuring the desired properties of post-issuance provider code. A question remains about the specifics of these guarantees when the card must share its processing with local and remote hosts, as it does, for example, in the SET protocol. There are many payment protocols other than the SET protocol, but the SET protocol is an interesting representative. Moreover, its specification is publically available and has become a benchmark for academic case studies. Hence the focus on SET here: other payment protocols may be more interesting commercially, but the technologies required will be comparable to those needed for SET.

Let us look now as the SET protocol with respect to the potential responsibilities of the embedded computer, that is, the smart card. The host device provides network connectivity and power and can carry out parts of the computation that are not considered sensitive. The terminal is likely to be chosen by

the user or the merchant, although it may sometimes be chosen by the issuer. Let us assume for the current discussion that the host consists of a PC and terminal chosen by the user and is trusted by the user, but that risk mitigation is desirable. For example, a corrupted PC should be unable to complete transactions if the card is not present in the reader and should not be able to confer the ability to make purchases to other machines. If the SET protocol is implemented entirely on the PC, then these guarantees cannot generally be made. If the PC has access to the keys that authorize transactions, then it can carry out transactions whenever it is attached to the network, and, if the physical security of the PC is lost, then the keys can be exported to other machines so they can make transactions. Given these risks, which of the steps in Figure 4 should be carried out on the smart card?

To see the challenge, consider the checking of certificates used to encrypt messages. In the PReq message the user (comprising both the host and embedded computer for now), includes information for the payment gateway such as the PANData, encrypted with the public key for the payment gateway. Suppose that the smart card is responsible for the signing and encrypting operations but checking of the certificate for the payment gateway is carried out by the host PC. The host PC could substitute a fake certificate so that the card put the PANData in a message encrypted under a key selected by the host, which could now obtain and store or distribute the PANData. It appears necessary to assign certificate checking to the card. However, checking certificates typically involves checking a chain of certificates starting from a root certificate and also checking revocation status by inspecting referenced Certificate Revocation Lists (CRLs). Full certificate checking is a complicated operation that requires significant memory and (if CRLs must be obtained remotely) network connectivity. Even assuming that the smart card could muster the space and computational capacity to do this, there is an more intrinsic problem: the lack of a clock with which to check expiration times.

Indeed it is general challenge regarding the use of a smart cards to ask which parts of the protocols can run on the terminal under various models of the trust level for the terminal. Work in [14] describes a general approach to this division based on a form of multi-level security. Sensitive values such as the private key for the card and the PAN are given high security level, while other data such as the OIData and XID are given low security level. The method partitions the code implementing the protocol into components that are assigned high and low values based on their treatment of sensitive values. High valued components must run on the card, whereas low valued components can be implemented on the terminal. There are also techniques to deal with the verification of certificates by the smart card. Lacking garbage collection, a Java Card is not able to do ordinary certificate checking, but it can engage the host in a sequence of messages that restrict the amount of memory the card needs to allocate at each step. Another idea is to entrust certificate checking to a remote control service, although this complicates the networking in transactions and raises questions about how the card knows what certificate is being checked given its inability to store and parse

the certificate itself. It is possible to check the certificate locally in pieces while using a remote service to confirm the time [14].

6 Refinement Architecture

We have now described a variety of technologies that could contribute to developing the concept of 'programmable payment cards' as sketched in Section 4. Smart cards provide the ability to embed some level of protected control in a host. Java Cards provide the ability to add post-issuance programs that might serve as approval policies. Such programs can be checked by the card using a digital signature, if the card supports the GlobalPlatform API, or verified on-card, if the card supports the assurance architecture in Figure 7. Sensitive steps of the protocol can be protected by the card while letting the host deal with less sensitive steps to help address card limitations. However, we still lack an overall model for reasoning about the security objective of a programmable payment card. The architecture in [14] provides a significant part of what is needed by modeling secrecy requirements. Secrecy is pivotal to controlling authorization, which is the main objective of the programmable payment card. However, it is not equivalent. For instance, a protocol that does not provide replay protection might allow an adversary to duplicate a transaction even if he is unable to decrypt the transaction messages.

The *refinement architecture* is based on the simple idea that the post-issuance programs on the card should always limit, and never expand, the sensitive transactions that a card is able to carry out. In a basic form, this is similar to a network firewall. Filtering firewalls examine packets and reject packets that match certain undesirable patterns. So, applying this concept to payment cards, we could impose a 'firewall' filter on the card that prevents undesirable messages from leaving the card. Communication units on smart cards are called Application Protocol Data Units (APDUs); they are classified into APDU commands and responses. However, it may make more sense to view communication units at a higher level. For instance, with a SET-based payment card, our primary attention may be on filtering PReq (purchase request) messages, or, more precisely, preventing the host from sending PReq messages that do not satisfy the policy of secondary provider but will be accepted as valid by the merchant and payment gateway. For example, suppose that merchants provide an OrderDesc (order description) that includes a service class such as whether the service is for accommodation or entertainment. Then the card filter could insist, for instance, that a PReq message will be created only with an OrderDesc for accommodation. This would refine the possible card events to include only acceptable accommodation purchases while eliminating unacceptable entertainment purchases. This approach makes at least two key assumptions that should be noted. First, message elements like OrderDesc must contain information on which policy can be based. If there is no service classification then it would be impossible for the card to make the necessary distinction. Second, the filter provider must trust that the merchant and payment gateway respect the payment protocol and the merchant provides

an honest OrderDesc. In particular, the user should not be able to collaborate with the merchant to formulate an order description that circumvents the filter. For instance, the merchant must insist on classifying entertainment as such even if it might cause a lost sale because of the policy filter on the card.

Is it possible to implement the refinement architecture using the filtering concept? Let us attempt a sketch of how this can be done assuming that all of the technologies in the previous section are at our disposal. We also need to assume some key trust relations. First, the user trusts the host he uses. Since this is likely to be his PC, this is a credible assumption. Second, the issuer and secondary provider trust the merchants and payment gateways as far as this is required for the SET protocol. Trust in the merchants includes trusting that OrderDesc provides an accurate description of the item being purchased. It does *not* include trusting the merchant to use the same PurchAmt with the payment gateway that it used with the user since this is already ensured by the SET protocol. Third, the card needs to protect its integrity against physical attack by the user, and against logical attacks from the host, merchant, and other parties to whom the card could be connected by a network link. A variety of additional assumptions would needed to prove the security claims formally, but these give a good start. The idea is to formulate the refinement as a conjunction of filters that are registered by secondary providers and invoked by the program that creates the PReq message. These filters are applied to the pair (OrderDesc, PurchAmt) together with auxiliary data such as the identity of the merchant and the time of purchase. Let us call this the *filter* refinement implementation.

The steps in the filter refinement implementation can be divided into installation steps by which a provider adds a filter, and transaction steps in which a user creates a purchase message. Messages for installation are described in Figure 8. The host of the provider loads and installs a CAP for an approval applet (filter) with the aid of the card manager. The approval applet has its own installation method, which is invoked by the card manager to initialize the approval applet. As part of this set-up, the approval applet obtains the Application IDentifier (AID) object of the transaction applet by providing a well-known reference value to the card manager. It uses this to access the Shared Interface Object (SIO) of the transaction applet. It invokes a registration method in this SIO to provide a reference that can be used by the transaction applet to later get the SIO of the approval applet. If this registration is successful then this is indicated to the approval applet, the card manager, and, finally, the host.

The steps in the transaction process are shown in Figure 9. When the host attempts to send a payment message, it needs to have the card create the signatures. To do this, the host passes purchase information D to the transaction applet on the card. This information includes OrderDesc, PurchAmt and other information. The transaction applet has a list of approval applets that it developed as these applets registered themselves. It now and then invokes each of these and provides them with D to get approvals. In the figure there are two approval applets that have been installed by providers. They act as filters on the information D. Only if both of them approve of D will the transaction applet

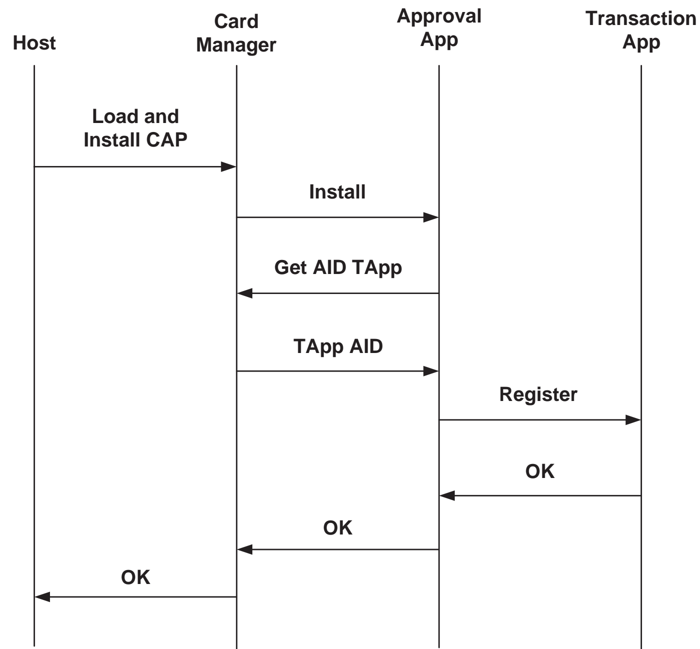

Fig. 8. Messages in Filter Refinement Installation

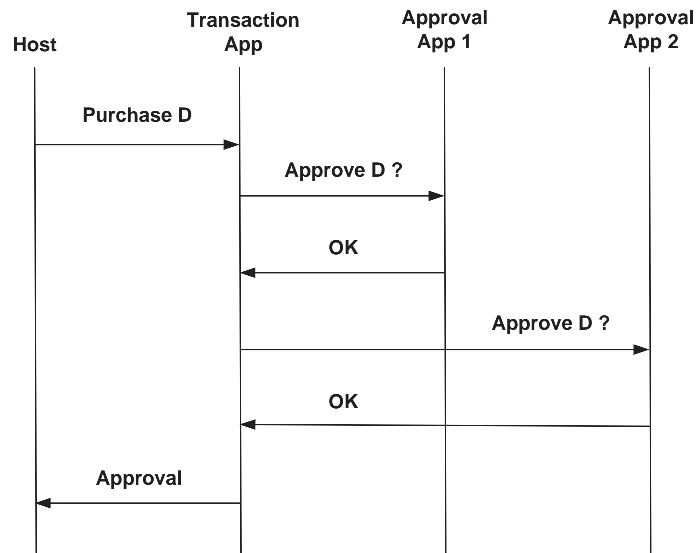

Fig. 9. Messages in Filter Refinement Transactions

create the signatures that the host needs to complete the purchase transaction. In particular, without the ability to create PReq, the user and host are unable to convince the merchant and the payment gateway to approve the purchase of the items in OrderDesc.

Now, what is needed to ensure the integrity of the filters? For instance, could a user install his own filter and use this filter to force approval of unauthorized purchases? The protocols in Figures 8 and 9 do not allow a filter to do more than *refine* the set of allowed events unless either (1) they have been given access to keys and other data necessary to complete transactions themselves or (2) they can corrupt the card memory to obtain these keys or otherwise trick the card into creating needed parts of PReq messages. The first of these is handled by choosing the programming interface for the card and transaction managers so that resources available to approval applets are limited to the purchase information they need. The second problem can be handled in either of two ways. In the GlobalPlatform approach, the approval applets can be inspected in advance by a certifying authority (Step 6 in Figure 6). The card will not load an approval applet without the necessary signatures. The certifier can perform checks such as showing that the CAP file is well-formed. In the method of Figure 7, the certifier is unnecessary and providers are able to place 'assurance transformed' CAP files on the card without needing a signature. The card then verifies such applets (Step 5 in Figure 7) and they are subsequently invoked by the transaction applet to obtain approvals.

There are some immediate questions that arise about the filter implementation. What happens if someone installs a bad filter that prevents the card from performing *any* transactions. Or, what if two filters interact to have this effect? We can assume that someone in physical possession of the card has the ability to destroy it, so the main threat arises if providers are are somehow able to gain unwelcome remote access to the card. Thus we must assume that whoever is in possession of the card uses a host that is capable of checking the credentials of a provider before allowing the provider to make an installation. Note that this is different from insisting that the *card* must be able to do this (as it does in the GlobalPlatform architecture). This can be viewed as an aspect of the requirement that the cardholder trusts his terminal.

Another question concerns how can a provider know what filters are already present to avoid redundancy? This raises also the question of what APDUs might be implemented by providers. It was argued above that we can prevent new APDUs from threatening the transactions that the card is able to make since the applet firewalls provided by the Java runtime will protect sensitive information. In the filter model as described above, the only SIO provided by the transaction applet is the one for registration, so there is a limited ability to usurp functions of the transaction applet. For convenience, filters could provide an APDU that allows them to be queried for some kind of description of what they filter. However, an applet with a rich set of APDUs (or SIOs) might risk its own security if the interface enabled tampering by some unexpected means.

The filter implementation has the advantage that it can be used without certifying or authenticating the filter code if on-card verification is possible. If it is possible or necessary to certify provider policies offline, then there are ways to go beyond the filter implementation and exploit technologies for proving refinement relations between policies. There is a considerable body of work on refinement, including automated systems linked to programming languages [5, 10, 24]. One example of a line of work on refinement [1, 2] breaks the process down into steps such as assume/guarantee reasoning, witness selection, and algorithmic state-space analysis to check transition invariants. Methods like execution monitoring [21] will be challenged by the size and time limitations of cards, but an executable monitoring language such as NERL [4] could be useful to define effective high-level monitoring.

7 Conclusion

A key challenge in the design of embedded control systems lies in deciding which functions should be performed in the embedded device and which should be performed remotely or locally in a more capable host. This tradeoff becomes more complex when the embedded control offers an open API. Open smart cards provide an early insight into specific challenges in this area since they have advanced to widespread recognition of the value of open APIs for embedded systems. Programmable payment cards offer a case study for architectural and assurance issues. Existing platforms and technologies, combined with the refinement architecture and filter implementation suggest that such cards are feasible even for comparatively complex transaction protocols. Platform support based on authenticated code could also enable the use of more advanced refinement techniques.

Acknowledgements

The work described here is based primarily on work in the OpEm project at Penn, so special thanks belong to the people participating in OpEm: Rajeev Alur, Watee Arsjamat, Alwyn Goodloe, Michael McDougall, and Jason Simas. OpEm is supported by ARO DAAD-19-01-1-0473, NSF CCR02-08990, and NSF EIA00-88028. I would like to thank David Tennenhouse for starting me thinking about programmable embedded devices, and Helen Gill for suggesting the idea of focusing on smart cards. I would also like to thank Fabio Massacci for helping me understand the SET protocol. Mykhailo Lyubich's work on SET for Java cards contributed in many ways to the OpEm project's understanding of the opportunities for an open API on payment cards.

References

1. R. Alur, R. Grosu, and B.-Y. Wang. Automated refinement checking for asynchronous processes. In *3rd International Conference on Formal Methods in Computer-Aided Design*, 2000.

2. R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *13th International Conference on Computer-Aided Verification*, 2001.
3. G. Bella, F. Massacci, and L. C. Paulson. The verification of an industrial payment protocol. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002. ACM Press.
4. K. Bhargavan and C. A. Gunter. Requirements for a network event recognition language. *Electronic Notes in Theoretical Computer Science*, 70(4), December 2002.
5. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Sax. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
6. D. Evans and R. I. Schmalensee. *Paying with Plastic*. MIT Press, 1999.
7. A. Goodloe, M. McDougall, R. Alur, and C. A. Gunter. Predictable programs in barcodes. In A. Jerraya and W. Wolf, editors, *Proceedings of CASES 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 298–303, Grenoble, France, October 2002. ACM Press.
8. G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FACADE: A typed intermediate language dedicated to smart cards. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering – ESEC/FSE'99*, number 1687, pages 476–493. Springer-Verlag, Berlin Germany, 1999.
9. C. A. Gunter. Micro mobile programs. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, pages 356–369, Montreal, Canada, August 2002. IFIP 17th World Computer Congress — TC1 Stream / International Conference on Theoretical Computer Science (TCS 2002), Kluwer.
10. G. N. K. Rustan M. Leino and J. B. Saxe. ESC/java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.
11. X. Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.
12. M. Lyubich. Eine SET Kundembörse mit der Java Card Unterstützung. In *GI Informatiktage 2000*. Konradin-Verlag, November 2000.
13. M. Lyubich. Die architekturen von SET mit der Java Card. In A. Bode and W. Karl, editors, *ITG Fachbericht, APC 2001 Arbeitsplatzcomputer*, 2001.
14. M. Lyubich. *Architectural Concepts for Java Card Running a Payment Protocol and Their Application in a SET Wallet*. PhD thesis, University of Rostock, 2003.
15. M. Lyubich and C. Cap. Eine implementierung von SET für Java. In *Tagesband Netzinfrustruckhur und Anwendugen für Informationsgesellschaft*, pages 208–214. Dr. Wilke Verlag, 1998.
16. Mastercard and Visa. *SET Secure Electronic Transaction Specification: Business Description*, May 1997.
17. Mastercard and Visa. *SET Secure Electronic Transaction Specification: External Interface Guide*, May 1997.
18. Mastercard and Visa. *SET Secure Electronic Transaction Specification: Formal Protocol Definition*, May 1997.
19. Mastercard and Visa. *SET Secure Electronic Transaction Specification: Programmer's Guide*, May 1997.
20. E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.
21. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3:30–50, 2000.

22. B. Schneier. *Secrets and Lies*. John Wiley & Sons, 2000.
23. Sun Microsystems. *Java 2 Platform Icro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000. White paper, http://java.sun.com/products/cldc/wp/KVMwp.pdf.
24. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *Springer LNCS*, pages 299–312, 2001.