Michael Baentsch, Peter Buhler, Thomas Eirich, Frank Höring, and Marcus Oestreicher
IBM Zurich Research Laboratory

JavaCard—From Hype to Reality

In this final of three related articles about smart card technology, the authors discuss the JavaCard, a much-hyped technology that is finally taking off as a multiapplication smart card.

In the smart card world, JavaCard has been one of the most hyped products around for years. The main reason for the hype is JavaCard's potential. Not only would it let all Java programmers develop smart card code, but such code could be downloaded to cards that have already been issued to customers. This flexibility and post-issuance functionality would significantly extend smart card possibilities. However, until very recently, such promises have not been backed by real implementations; JavaCard existence has been limited to reference implementations—better known as simulations. However, JavaCard implementations now exist and the technology is beginning to live up to the hype.

A JavaCard is a typical smart card: it conforms to all smart card standards and thus requires no change to existing smart card-aware applications. However, JavaCard has a twist that makes it unique: a Java Virtual Machine is implemented in its read-only memory (ROM) mask. The JVM controls the access to all smart card resources, such as memory and I/O, and thus essentially serves as the smart card's operating system. The JVM executes a Java bytecode subset on the smart card, ultimately providing the functions accessible from outside, such as signature, log-in, and loyalty applications.

The advantages of this are obvious: Instead of programming the card's code in hardware-specific assembler code, new applications can be developed in portable Java. Moreover, applications can be securely loaded to the card post-issuance—after it's been issued to the customer. This lets vendors enhance JavaCards with new functions over time. For example, banking cards that initially give customers secure Internet access to their bank accounts might be upgraded to include e-cash, frequent flier miles, and e-mail certificates (see Figure 1).

The main reason it took three years to get real, interoperable cards into developers hands was a knowledge chasm between the developers of Java and those who understood the resource-constrained smart card environment. A primary discussion area was how to shrink Java to make it fit on a smart card. The chasm closed only recently, with the advent of real implementations and their deployment in customer scenarios. Competitive technologies, such as the Windows card, have also helped to bring together the different entities working on the JavaCard specification. Real JavaCards are now being produced in volume.

Here we outline the JavaCard basics, including its history and technology, and describe our experience shaping JavaCard technology.

Background

The basic idea of the JavaCard is not new. Starting in 1996, Schlumberger, a smart card manufacturer, demonstrated a

1092-3063/99/$10.00 © 1999 IEEE

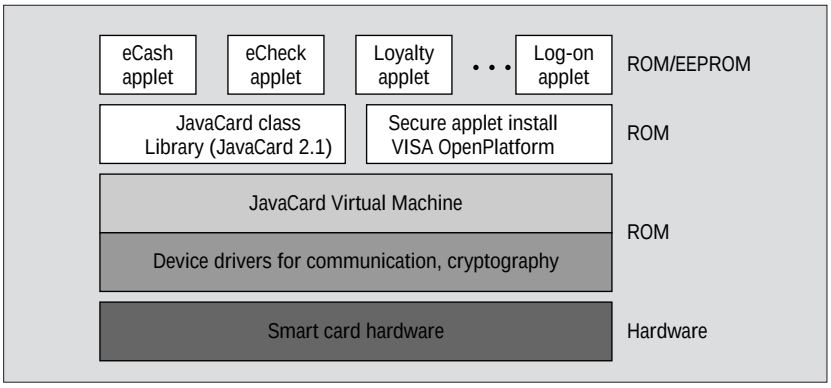

Figure 1. JavaCard includes a Java Virtual Machine, which lets vendors add new applications even after the card has been issued to customers.

Java-based smart card by adding a lightweight Java bytecode interpreter to a smart card's OS and downloading Java class files, which were converted to a smaller, proprietary format. The original OS functionality was used to download, manage, and execute the application code and its data. There were at least two major drawbacks to this initial effort. First, a Java application could only access a few system methods, which was insufficient to build a card that anyone could program. Second, it had to rely on the underlying card's file system to load and store data persistently—an approach clearly unsuitable for an object-oriented runtime system.

Evolving standards

Even though it was still unclear what the platform-neutral abstractions of a Java smart card should look like, Sun Microsystems issued a first JavaCard specification in October 1996 (java.sun.com/products/javacard), based on Schlumberger's experience. The specification limited its description to the JavaCard's general goals and architecture: it should provide support for a Java-language subset and offer APIs for smart card-specific functions like cryptographic operations.

In February 1997, GemPlus and Schlumberger founded the JavaCard Forum (www.javacardforum.org). Smart card manufacturers such as De La Rue, Bull, and Gieseke & Devrient (G&D) joined the forum and released a new specification at the end of 1997. This JavaCard 2.0 specification contained more concrete details on both API specifications—such as those for accessing the underlying memory and cryptographic functions—and the JavaCard Virtual Machine (JCVM), a smaller and simpler JVM that sheds many of standard Java's features, as we describe below.

However, portability and interoperability of JavaCard code (also known as *applets*), among different JavaCards was still nonexistent, even on a source code level. For example, the file system API specification was not generic enough to accommodate various vendors' proprietary file systems, and the crypto API lacked functionality and was not flexible enough to enforce the export restrictions of different countries. Even worse, the class file conversion, download process, and executable instructions were still up to the JavaCard implementor. Finally, because the API specifications lacked detail, a reasonably complex JavaCard application was impossible to port from one card to another.

Visa's OpenPlatform

To move the multiapplication smart card area forward, Visa introduced the Visa OpenPlatform smart card specification in April 1998 (www.visa.com/nt/suppliers/open/main.html).[1] VOP defined an architecture for managing applications on multiapplication smart cards—that is, it specified standards for securely download and installing new applications.

From the start, VOP was geared toward the JavaCard, going as far as to define on-card Java APIs for accessing the security controlling objects from within applets. In 1999, Visa opened the standard, renaming it *OpenPlatform 2.0* (OP) and relinquishing sole control. This let the ETSI standardization committee, for example, choose OP for managing SIM cards in GSM mobile phones.

Given that neither Sun nor the JavaCard Forum had established an open standard for secure applet download, Visa's OP has become a de facto standard for managing multiapplication smart cards; currently, it is all but mandatory that vendors implement JavaCard and OP on the same chip.

However, OP does not define the JavaCard application code's file format or instruction set, as this was still under discussion throughout 1998. Such questions about a standard for JavaCard bytecode touched upon one of the main selling points of JavaCard at that time, namely the emphasis on Java's underlying security architecture, including the JCVM's bytecode verification. Although no suggestions were forthcoming as to how to implement this in the resource-restricted smart card environment, many JavaCard vendors considered on-card verification capabilities necessary; they wanted future cards to contain all information required for on-card bytecode verification—regardless of the practical consequences, such as the inability to actually implement such features on the cards.

JavaCard 2.1

The JavaCard 2.1. specification (JC21), released in March 1999, offers JavaCard bytecode verification as an optional feature supported by the load-file format, but favors a different security model, known in the PC world as *code signing*. Using the secure download process as specified in OP, applet code is run through a secure converter and evaluated, then signed with the card-issuer's secret key prior to downloading. Primarily for marketing reasons, JC21 does not claim to completely trust this external verification and the enforced Java language safety mechanisms known as *sandboxes*.

Therefore, JC21 introduces an additional concept: a software firewall mechanism unique in the Java world. Objects are explicitly associated with their owning applets, and several additional checks in the JVM must verify proper access rights on each object access. The file system API has been dropped completely due to the huge differences between different established file system formats. The cryptographic API has been extended to numerous classes.

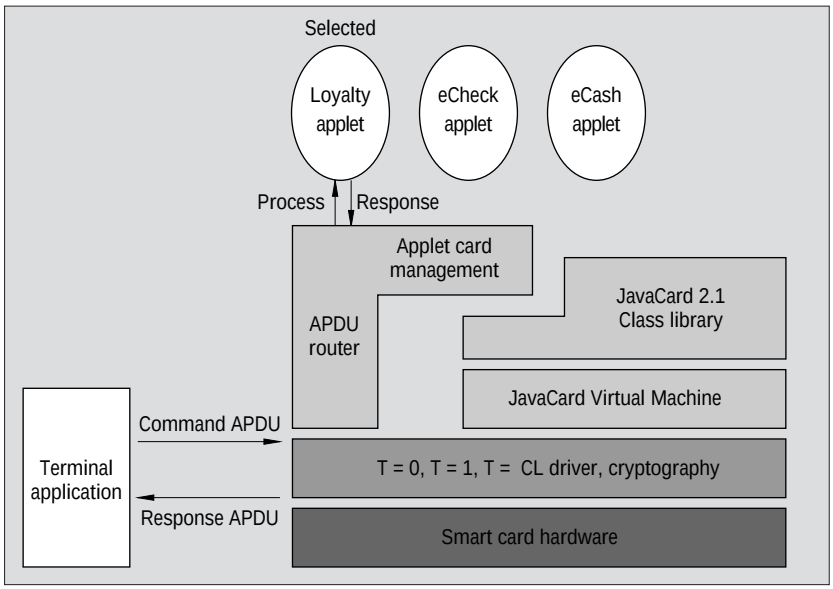

Figure 2. The JavaCard has a simple, server-like operation mode. Once a command arrives, it is decoded and either selects a particular applet or is forwarded to a preselected applet, passing control to the Java code. (APDU = application data protocol units.)

Current issues

The end point of the JavaCard specification process has yet to be reached; more and more standard Java concepts are now finding their way into JavaCard Forum discussions. To highlight the difficulties this raises, we briefly recap a smart card's underlying hardware constraints.

An inexpensive (and thus mass-marketable) smart card chip has 256 Bytes of random-access memory (RAM), 16 KBytes of ROM, and 4-8 KBytes of electrically erasable, programmable ROM (EEPROM). The Rolls Royce of smart card hardware currently has 4 KBytes of RAM, 64 KBytes of ROM, and 64 KBytes of EEPROM. The "middle class" smart card hardware falls somewhere in between, and differs mostly in whether it provides a cryptographic coprocessor for RSA and DES encryption.

Given that the cost of a single smart card—and thus a JavaCard built on it—is critical because of the number of cards issued, the lower end of the hardware spectrum is the most interesting for a JavaCard implementation. To accommodate the hardware limitations, Sun and JavaCard Forum have defined several restrictions of standard Java. For example, several data types (including Float, Double, and Long) are not supported; the bytecode is slightly different; and threading is not supported. Such limitations do not seriously affect the ability to write OO programs or applets for the card, primarily because the JavaCard has a simple, server-like operation mode, which we will now describe.

JavaCard basics

JavaCard operates like a typical smart card: When the smart card reader sends a command, the JavaCard processes it and returns an answer (see Figure 2). To maintain compatibility with existing applications for smart cards, a single JavaCard can process only one command at a time.

Internal operations

The JavaCard's internal JVM boots up when a smart card terminal activates the hardware and returns the Answer To Reset (ATR), indicating its communications capabilities. The system then enters a loop, waiting for commands from the smart card terminal. When a command arrives, it is decoded and either activates (*selects* a particular applet) or is forwarded to a previously selected applet. In the latter case, control passes to the Java code, invoking the applet's `process()` method. The JVM then executes any operations sequence programmed into the applet. This might include reading further data from the smart card terminal, decrypting some data using the JavaCard Cryptography API calls, or storing data in objects allocated in the smart card's EEPROM area. Once the method terminates, the JavaCard's runtime environment returns a status word to the terminal, along with all the data the applet has written to the I/O interface.

During bytecode execution, JavaCard honors all critical Java concepts. In terms of bytecode execution security, the Java Card enforces known runtime Java virtues. For example, illegal array accesses are prohibited and type-safe object assignments are ensured. On the programming level, JavaCard includes support for class hierarchies, instance-of relationships, and exceptions. In this respect, the JavaCard is not different from standard Java. However, in other areas, JavaCard is distinctly different.

The differences are clearly visible on the application level. One example is in transaction safety, which must be accounted for by both applet programmers and JCVM designers. Because the JavaCard can always be extracted from the power-supplying smart card terminal, the applet's internal data structures must change in a logically atomic manner—that is, all-or-nothing. The JavaCard specification provides a specific API for this problem, and the JCVM design must take this into account—by checking a transaction recovery buffer each time it boots, for example.

Another difference is that a smart card can access RAM and EEPROM. RAM is fast, small, and nonpersistent, and EEPROM is slow, big, and persistent. The JavaCard APIs visibly distinguish between the two memory types at the application level so that programmers can develop applets to perform at a level comparable to traditional smart cards programmed in hardware-specific C or assembler code.

Implementation

JavaCard applets can be implemented using familiar Java development environments such as VisualCafe or Java Developer's Kit. Such environments often offer a JavaCard simulator, which lets applet developers concentrate on application functionality and the correct API usage. To this end, the applet is run in a typical desktop JVM but accessed only over the smart card-specific communications interface. Thus, code that will later interact with the card's

applet can be tested during applet development. In practice, however, the simulation environment is of dubious value as it often tempts developers to disregard the constraints of a real smart card.

In integrated development environments, the only visible differences between applet development for the PC and the JavaCard are the particular set of API calls for JavaCard applets and a post-processing stage, which is required to adapt the Java bytecode to the resource-constrained smart card environment. To this end, after compiling the applet's code, a converter is run on the Java class files preparing them for download (see Figure 3). Conversion removes a significant amount of data that is necessary only for supporting the dynamic download of new classes at runtime. The JavaCard platform has no need to support such runtime downloads as a smart card's primary purpose is to obviate the need for online connectivity.

In place of dynamic class download, the JavaCard specification prescribes a closed-package concept in which all classes of one package—either a library or application—are downloaded to a card at the same time. To accomplish this, classes are packaged and converted into a JavaCard CardletPackage (CAP) file. In this step, the off-chip converter removes the space-intensive link information of a typical Java class file: the field, method, and class names. Thus, a standard Java class file created by the initial compilation step must first be parsed and converted into an internal representation. It can then be bound against referenced classes, which might cause the other classes to load and bind. For more detail on conversion techniques, see the sidebar, "Optimizing conversion."

Creating such linkable JavaCard code images raises a second issue: Java class initialization code must be executed in standard Java after it has been loaded and bound, but before its code or elements are initially accessed. The JavaCard environment therefore must simplify the load, bind, and initialization model, which is accomplished by removing the requirement for class initialization at runtime and prescribing a one-step execution for downloading and linking all classes.

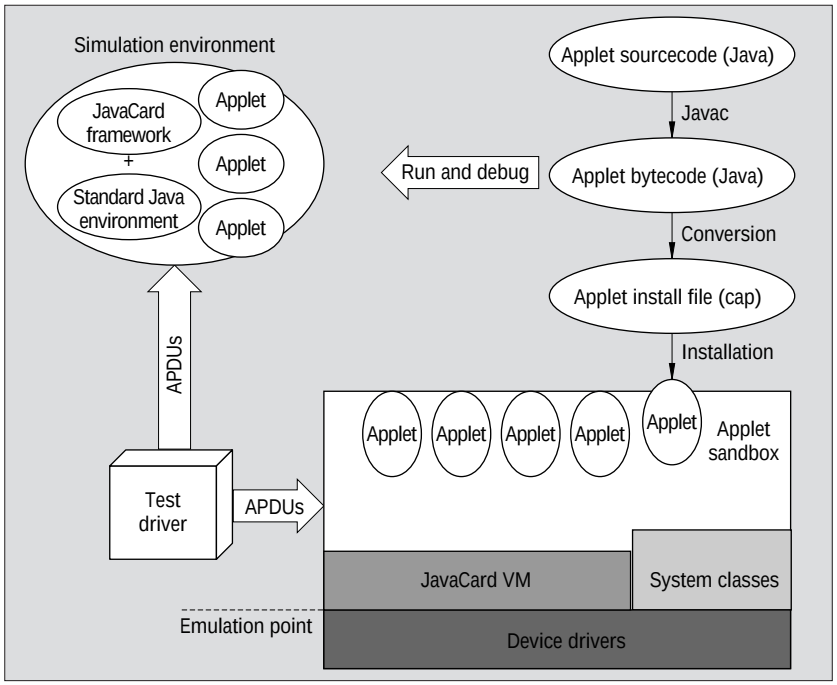

Figure 3. Applet development and lifecycle. Once applet code is compiled and debugged, a converter removes data needed for dynamic class download, which is not required in the JavaCard. Converted applets are then installed on the JavaCard.

Following these off-card preparations in the development environment, an automated, combined load-link procedure is executed on the card. If all classes are perfectly known at convert time and a fixed JavaCard code image is desired, this step can be omitted. However, this is typically the case only for the system classes going into the JavaCard's ROM image, such as the JavaCard API or OP classes. With post-issuance applet upload, this technique cannot be applied because each JavaCard might have its own system class implementation in different physical locations. The converter would thus need a detailed knowledge of the destination card, the system-class implementation and version, the object layout, and so on. Besides the logistical problem, this information is unavailable in disconnected applet downloads, which is one of the JavaCard environment's main goals.

With online downloads, where minimizing the size of on-card code has a higher priority than disconnected applet download, omitting the link process is possible. However, because it is questionable whether the size benefit justifies a total departure from the original Java idea—to load new functionality—online download and related concepts are not prescribed for the JavaCard.

OpenPlatform architecture

As we discussed, the JavaCard specification does not define how to load preprocessed applets into the card. All actual communication formats as well as security-related procedures for this are defined by OP.

The OP security concept focuses on the defined roles of the different software providers for multiapplication cards:

- the card's OS provider (the entity that implements the JavaCard),
- the card manufacturer (the entity that physically embeds the smart-card chip and personalizes the card on a logical level),
- the card issuer (the entity offering the cards to its customers), and
- the applet provider (the entity providing post-issuance functionality).

The goal of OP is to separate concerns. Once the cards leave the manufacturer's premises, for example, only the card issuer should be able to control the card's behavior and contents. This becomes particularly important in a legal sense, once the cards have been issued to customers.

To this end, OP introduced *CardDomain*, an on-card abstraction representing the card issuer that uses cryptography

Optimizing conversion

There are two basic techniques for optimizing the JavaCard byte code set in a conversion step executed prior to downloading applets to the card. The first technique is to optimize the instruction set; the second, to optimize the overall data layout.

Instruction set optimization

Changing the standard Java instruction set alone may not be sufficient to obtaining an efficient, high-performance JavaCard instruction set, but it is a valid way to achieve better overall JavaCard system performance. The first step is to remove all instructions operating on unsupported data types. You then reorder the remaining Java operation codes (opcodes) to suit an efficient JavaCard Virtual Machine implementation. Next, because the JavaCard's default supports only short integer arithmetic, you define and use a short integer instruction instead of the standard Java integer instruction set. This limits the stack operands to short integers and thus saves stack space at runtime. Thus, the optional integer arithmetic in the JavaCard instruction set always operates on multiple stack slots similar to the long instruction set in standard Java.

You can obtain additional code savings by limiting the argument size of various instructions. For example, the table jump instructions are only given non-aligned short integer arguments. You may also introduce a new JavaCard instruction for instruction sequences that occur often in standard Java bytecode. For example, JavaCard 2.1 introduces a new opcode that loads a field of the object referenced by the first local variable (usually the reference to "this"). This avoids the opcode sequence of first loading the local variable on the operand stack and then accessing the field using the operand pushed onto the stack.

Data layout

When defining a JavaCard file format, structuring and laying out the linking information is a complex task. The goal is to provide a format that optimizes the timing and memory limitations of smart card hardware.

Time limitations require minimizing the size of output files that will be downloaded over the slow smart card communications link, as well as the number of writes to the persistent, but slow EEPROM memory required during code download and applet installation.

Memory limitations require minimizing the RAM-based temporary resources needed to download the JavaCard CAP file and the EEPROM-based applet execution information, which cannot be discarded.

The first problem is in deciding how to map the symbolic information in the standard class files to short identifiers for their classes, members, and fields. Because a global name space is difficult to manage, each identifier should only be valid relative to its defining package. In JC21, the identifiers for given classes and class members are indices or offsets into class or method tables in the destination package.

The name mapping for a published API is part of the API specification and is distributed as an export file. This gives developers the API so they can correctly convert packages on any converter. The export file contains only information about the package's externally visible items, but includes additional information like a method's access right or the values of final static fields.

The second problem is in deciding

to control access to the card's functionality. Thus, the owner of the CardDomain's keys ultimately determines such things as whether new applets can be downloaded, which applets are active at any one time, or whether a card is altogether disabled.

OP also accommodates different applet providers on the same card. For example, an airline might provide loyalty applets—such as frequent flyer miles—to a card typically designated for banking applications. OP facilitates such cohosting with an on-card abstraction called *SecurityDomain*. Comparable to the CardDomain, SecurityDomain keys are in the sole possession of an authorized applet provider. Thus, not even the card issuer may know the actual code of the applet.

The code's good behavior on a card is guaranteed in part by business agreements, as well as by the technical provisions build into the JCVM interpreter and the off-chip converter described above. Installing SecurityDomain is the sole responsibility of the card issuer and is technically controlled through knowledge of the correct CardDomain keys.

Through these two basic means, OP clearly enforces the card issuer's sole responsibility for a card's fate. No other entity involved in developing a multiapplication card and its code can manipulate cards once they are in customers' hands. Moreover, OP is not only bound to the JavaCard, but has been defined on different abstraction levels. At the highest level are language-specific on-card APIs, which need minor adaptations to work on other multiapplication cards, like the Microsoft Windows Card. At the lowest abstraction level are communication level commands, which are fixed in the specification to guarantee interoperability.

OP's influence on JavaCard development tools is minimal. In development, a program is required to transform the JavaCard CAP files into (possibly encrypted or signed) smart card communication command sequences (application data protocol units, or APDUs). These are sent through a standard smart card terminal to the JavaCard, where the CardDomain accepts the data, and loads and links the new applet into the card. In addition to such a download tool, further programs can execute general OP tasks, like activating or removing an applet, installing SecurityDomain, or blocking a card for use. In a GUI-driven development environment for JavaCards, tools such as those for Java class file conversion, static safety checks, applet signing, and download can be neatly hidden behind one 'Download Applet' button.

JavaCard architecture

With a JavaCard conforming to JC21, the lowest level of the card's ROM code consists of memory access (RAM, ROM, and EEPROM) and device drivers for the I/O. Drivers for accessing the cryptocoprocessor are sometimes required. Such drivers are collectively used by the assembler or C implementations of the native methods of the respective API calls. An example of this is `sendBytes()` in the APDU class, which manages smart card communications behavior over APDUs for all JavaCard applets.

A significant part of the overall ROM code already consists of JavaCard byte-

how to efficiently organize the file format so that locations containing identifiers to be bound are easy to find. One way to achieve this is to expect the card's loader to parse the CAP file, find the identifiers, and bind them. Another way is to provide a table for the location needing a fix-up.

The JavaCard specification uses both approaches. The card's loader, for example, is expected to load the class descriptors on the fly, bind them, and write them into the persistent memory. For the various bytecode instructions that reference classes or class members, the second scheme is used: A CAP-file table references the various instructions, which in turn refer to items in the constant pool that contain target identifiers.

This combined scheme is quite difficult and expensive to implement, but it does let you discard the constant pool after instructions are bound. However, this technique also requires that you load the instructions before you load the location table, and apply binding on the fly—that is, as the download commands arrive piecemeal in the card. Unfortunately, the persistent memory, which ultimately contains the instructions, must therefore be written twice, leading to a less-than-optimal performance during applet upload. We have shown in our implementation that it is also possible to base the whole file format on an on-the-fly binding scheme, with only one EEPROM access per memory location during the download.

In any case, the CAP file must contain as much pre-resolved information as possible to save expensive card resources. Because the converter leaves out information from the original class files, you must ensure that the CAP file contains all the necessary execution information as the card cannot recompute them at runtime. JC21, for example, distinguishes internal and exported items, which permits a faster binding on the card. Also, link information for applications that are not a link target later, is separated and can be eliminated.

The converter should also sort the fields of references and primitive types to permit an efficient garbage-collection scheme—that is, to let the JavaCard reclaim objects no longer in use for future memory allocations. The converter can save additional space by compacting the various class and method descriptors. For example, it can reduce attribute size, such as the number of local variables, or leave out unnecessary information, such as access attributes. These can then be converted into a form that is close to the execution layout. This technique for improving runtime-efficiency is comparable to that used in standard Java's Just-in-Time compiler.

Further options

Because of the JavaCard's closed-package concept, the converter can also apply more sophisticated optimizations. These include, for example, inlining methods—typically constructors or simple accessor methods that might then be left out of the resulting CAP file. As another example, a class hierarchy analysis offers the opportunity to replace virtual method invocations by direct invocations and to compact method tables. This results in an overall performance improvement and an overall code size reduction. However, these optimizations can change the Java semantics and might make it impossible for an external application to verify the CAP file without having access to the original Java class files. As external CAP-file verifiers are announced, such optimization options will have to be considered carefully.

code implementing the various APIs defined in the JavaCard specification. As we discussed earlier, the OP functionality must be provided in ROM as well if card control that conforms to multiapplication standards is desired. Because OP requires DES cryptography, an impressive list of functions must be implemented and—most importantly—must fit into the 20 KBytes of a simple smart card.

Real-world JavaCard

We have built two JavaCards: one conforming to the lower end of the hardware-capability and price spectrum and the other in the middle, with 32 KBytes ROM and 1200 Bytes of RAM. To test the cards, we measured real-world applets running on them. Our results show that a functioning JavaCard can be built if the right abstractions and APIs are chosen.

As a reference application, we implemented Visa's e-cash protocol, *VisaCash*, as a JavaCard applet. The applet has a code size of 5 KBytes and performs all VisaCash operations, including the loading and spending of electronic money secured by DES and triple-DES cryptography; the storing and retrieving of logfile information; and Visa's mandatory, cryptographically secure personalization procedure. The VisaCash applet is certified by Visa to operate in total accordance to specifications and has been rolled out to customers.

For a baseline comparison, we tested our "middle class" card against a dedicated, conventional VisaCash card, which is in production use; and a card using another implementation of the JavaCard specification.

Results

We programmed our card in smart card specific native code, following to the greatest extent possible the intermediate JavaCard specification of mid-1998. The complete virtual machine code is written in C, the device drivers are assembly, and the OpenPlatform implementation is mostly Java. We took all measurements on the same smart card terminal type, thus guaranteeing that identical clock rates drove the cards during the experiments. The JavaCards ran on identical hardware platforms. Table 1 shows the results for the three most typical operations: *Select* application, *Load* money, and *Purchase* goods.

Discussion

Provided that card APIs are perfectly tuned to the specifics of the resource-limited smart card environment, our experience shows that the JavaCard can run at performance levels akin to conventional cards. This is primarily due to the fact that the most time-consuming operations are written in assembly code. These operations include the cryptographic operations, communications, and EEPROM writing. The reasons for the Reference JavaCard's poor performance on the identical hardware can only be guessed—it's possible that developers used the wrong APIs or that it was simply a bad implementation, or that it was some combination of these and other factors.

We have submitted solutions to the two most important issues for a high-perfor-

Table 1. VisaCash performance benchmark (numbers in milliseconds).

	VisaCash Card	IBM JavaCard	Reference JavaCard
Select	24.6	15.0	107.3
Load	203.4	220.2	932.8
Purchase	207.9	217.8	823.9

mance JavaCard implementation to the JavaCard Forum, Sun, and scientific conferences. The first issue is the need to efficiently handle transient memory (RAM)[2] and will soon appear as an amended JavaCard specification version. The second issue, transaction support, is still under consideration (www.javacardforum.org).

Our JavaCard implementation experiences are also telling in the impact of various optional features in the JavaCard specification. Table 2 shows rough estimates of the ROM code size necessary to provide the different components of a fully functional OP JavaCard. We implemented a JavaCard with full public-key support on a "middle-class" smart card with about 1 KByte RAM and 32 KBytes ROM. Depending on a developer's skill, it may or may not be possible to include—as we did—an optional Java garbage collector, public key cryptography support (signing, encrypting/decrypting/hashing data, generating keys on the card), or further applets of general value, such as PKCS#11-type data stores.

In the end, the market will ultimately decide which features will be required from a JavaCard, and how much they should cost—that is, how big and expensive the JavaCard's smart card chip will be. The comparison with other multiapplication smart card technologies, like the Multos or Windows cards will drive this development.

Applications

The proof-of-concept software we developed over two years during the constant redefinition of the JavaCard standard has proven valuable in several field pilots, including the JavaCard installation at the US General Services Administration (policyworks.gov/org/main/me/smartgov/information/holcombe_pp0599/sld018.htm). At the US GSA, the JavaCard provides a biometric-driven secure login, Netscape e-mail signing and encryption, as well as mundane tasks such as storing frequent-flier miles.

Although our card is not fully compliant with the March 1999 JC21 specification, our experiences did heavily influence the JavaCard specification process. We are now in the process of having our software commercialized—under the GemXpresso brand name—through a licensing agreement by GemPlus, the world's leading smart card manufacturer. Other, fully JC21 compliant cards are expected by the end of this year from Schlumberger, DeLaRue, and Bull.

JAVACARD SPECIFICATIONS have significantly improved since the hype began three years ago. The time has now come to see how well different implementations will interoperate given the difference in functionality and options for improvements we discussed here.

Ultimately, performance, price, and interoperability will be the litmus test for JavaCard's success in the real world. If applet developers can write their own code for one particular, inexpensive card and run it unchanged on any other JC21 compliant card, the JavaCard hype will finally be a reality you can buy and rely on.

Table 2. JavaCard component sizes in kilobytes.

Virtual machine	4.0
Memory management subsystem (including transaction support)	4.0
Garbage collector for RAM and EEPROM	1.5
DES implementation (no hardware)	1.4
RSA/DSA implementation (PK coprocessor)	2.4
On-card RSA/DSA private-key generation	0.6
PK hash algorithm (SHA1)	0.6
T = 1 protocol	1.0
T = 0 protocol	0.5
T = CL (contactless) protocol	0.5
JavaCard system classes (no crypto)	2.5
JavaCard crypto classes (IBM proposal)	0.7
JavaCard crypto classes (JC21)	ca. 5.0
OP implementation (mixed native/Java code)	8.0
Full-fledged SecurityDomain support	1.0
Applets required to be in ROM for VOP compliance	1.2

Acknowledgments

We thank VISA International, and in particular, Gavin Shenker of Visa's OpenPlatform team for his input to our work, which led to many fruitful discussions within the JavaCard community. At IBM, we thank our former manager, Francois Dolivo, for his diligence in pushing through our work. Finally, we thank Eric Alzai's JavaCard technology evaluation team at GemPlus for preparing the decision to license our work and making it the basis of GemPlus' JavaCard products.

References

1. M. Oestreicher and K. Ksheeradbhi, "Object Lifetimes in JavaCard," *Proc. Usenix Workshop Smart Card Technology*, Usenix Assoc., Berkeley, Calif., 1999, pp. 129–137.
2. M. Oestreicher, "Transactions in JavaCard," *Proc. Annual Computer Security Applications Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., to appear, Dec. 1999.

Michael Baentsch is a research staff member in the Secure Systems Group at IBM Zurich Research Laboratory. His interests include secure hardware tokens, as well as practical aspects of security, such as protocols or applications requiring cryptographic functionality. He received a PhD in computer science at the University of Kaiserslautern, Germany, and is a member of the ACM. Contact him at baentsch@acm.org.

Peter Buhler is manager of the Secure Systems Group at IBM Zurich Research Laboratory. His interests include cryptography, secure protocols, and embedded systems. He received a PhD in computer science at the University of Kaiserslautern, Germany. Contact him at bup@zurich.ibm.com.

Thomas Eirich is a research staff member in the Secure Systems Group at IBM Zurich Research Laboratory, where he develops applications for cryptographic algorithms, such as advanced software distribution mechanisms. Eirich received a PhD in computer science at the University of Erlangen, Germany. Contact him at eir@zurich.ibm.com.

Frank Höring is a research staff member and software engineer in the Secure Systems Group at IBM Zurich Research Laboratory. His interests include cryptographic APIs and PKCS#11 security tokens, as well as low-level operating system functionality. He received an MS in computer science at the University of Erlangen-Nuremberg, Germany. Contact him at fhr@zurich.ibm.com.

Marcus Oestreicher is a PhD student in the Secure Systems Group at IBM Zurich Research Laboratory. His interests are in advanced virtual-machine design and bytecode optimization techniques. He received an MS in Computer-Science at the University of Erlangen, Germany. Contact him at oes@zurich.ibm.com.