Paper VI: Secure Networked J2ME Applications: Problems and Challenges

Secure Networked J2ME Applications: Problems and Challenges

André N. Klingsheim, Vebjørn Moen, and Kjell J. Hole

Abstract

An increasing number of smartphones support the Java 2, Micro Edition (J2ME) platform. The authors discuss problems and challenges of writing secure client-server applications for these phones. In particular, they explore the security of the current J2ME platform, and examine the new Security and Trust Services API for J2ME.

1 Introduction

A smartphone is a high-end mobile phone featuring a large color display and more processing power than regular mobile phones. A key feature is the ability to install additional applications. The smartphone market is growing fast [1]. The increasing availability of smartphones stimulates the market for rich content such as games, news, media (audio/video), and adult content. Games for mobile phones have been around for several years, but lately we've seen a rapid growth also in this market. Total global revenues in the mobile gaming market were around 2.6 billion USD in 2005, and is estimated to increase to 11.2 billion USD by 2010. It is also anticipated that 20.5% of the global revenues in 2010 are generated by online multi-player games [2].

Many different development platforms exist for smartphones, categorized by phone manufacturers, mobile operating systems and device capabilities. The most widespread platform is Java 2, Micro Edition (J2ME, see `java.sun.com/j2me`), available on nearly 80% of the smartphones currently on the market.

In this article, the authors discuss experiences gained during a commercial J2ME development project. They discuss how J2ME technologies are implemented on real devices and provide insights into the problems and challenges that occurred during the development process. Current and future J2ME security related functionality is outlined. The reader should be able to make educated decisions on how to develop secure J2ME applications after reading this paper.

The rest of the article is organized as follows. Section 2 gives an overview of the J2ME technologies, Section 3 discusses some of the problems and challenges on the mobile platform, and Section 4 outlines the current J2ME security toolbox. Section 5 discusses how a malicious J2ME client can attack the server, Section 6 gives an overview of the new Security And Trust Services API (SATSA), and Section 7 concludes the paper.

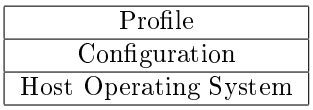

Figure 1: J2ME Architecture

2 J2ME technologies

The Java platform has several editions, the reader may be familiar with Java 2, Enterprise Edition (J2EE) for the server side and Java 2, Standard Edition (J2SE) for desktop systems. J2ME is a highly optimized Java runtime environment aimed at mobile phones, Personal Digital Assistants (PDAs), and other small devices. J2ME introduces *configurations* and *profiles*, see Figure 1. Configurations are intended for devices with similar characteristics in terms of processors and memory. Profiles, on the other hand, target devices that are similar in terms of screen type, input devices and network connectivity, and complement the low-level functionality of configurations by adding support for e.g. user interface and network connectivity.

A configuration specifies the Java Virtual Machine (JVM) features supported, the included Java programming language features, and supported Java libraries and Application Programming Interfaces (APIs). Two configurations are widely available on J2ME enabled devices, Connected Limited Device Configuration (CLDC) versions 1.0 and 1.1. Version 1.1 is deployed in newly released J2ME devices, while version 1.0 has been around for years. The most notable difference between the two is that CLDC 1.1 adds support for floating point operations. Since CLDC 1.1 is a superset of CLDC 1.0, Java applications built for CLDC 1.0 will run without problems on CLDC 1.1. Unless any of the additions in version 1.1 is strictly needed it is therefore recommended that developers build their applications for CLDC 1.0 to be compatible with as many devices as possible.

Two profiles exist that extend the functionality of CLDC, namely Mobile Information Device Profile (MIDP) versions 1.0 and 2.0 [3]. The MIDP profiles add APIs for user interface, network connectivity and persistent storage. Network connectivity is provided through HTTP connections, and persistent storage is provided through a record management system. MIDP 2.0 is a superset of MIDP 1.0, and includes support for secure HTTP (HTTPS) connections and more powerful graphics APIs for gaming.

New J2EE/J2SE/J2ME components are developed through the Java Community Process (JCP) program. Java technology specifications are created by first submitting a Java Specification Request (JSR) that must be accepted by an executive committee. Once a JSR is accepted, an expert group is formed to take responsibility for the specification development. The specification draft must first pass a community review, and then a public review. Finally, a proposed final draft is presented, and a final approval ballot decides if the specification is suitable for a final release.

An expert group typically consists of many parties. As an example, the MIDP 2.0 expert group included, amongst others, Ericsson Inc., Motorola, Nokia, Siemens, Sun Microsystems Inc., Samsung, and Symbian Ltd. The formation of an expert group enables large industrial parties to collaborate in order

to specify new functionality for the Java platform. Hence, competition between different vendor specifications is minimized, increasing the likelihood of wide adoption of a specification. All CLDC versions and MIDP versions are results of JSRs, as well as several other J2ME components such as SATSA (JSR-177), the Wireless Messaging API (JSR-120), and the Java APIs for Bluetooth Wireless Technology (JSR-82). All specifications are freely available at the JCP website (`www.jcp.org`).

3 Current problems and challenges

J2ME developers face several problems and challenges. Our experience with J2ME enabled smartphones is that the implementations generally have some quality issues. Developers are likely to spend time solving problems that occur because of the varying quality of the J2ME implementations. Specific phone models can have their own bugs, forcing developers to maintain several parallel versions of their source code to support as many devices as possible. This situation is not consistent with the Java philosophy of "Write Once, Run Anywhere," and severely increases the complexity of mobile software development and maintenance. In our experience, MIDP 2.0 implementations have less bugs than MIDP 1.0 implementations. However, to cover as much of the market as possible developers have to consider all the existing MIDP 1.0 devices already out there. MIDP 2.0 devices still constitute a minority of all J2ME devices sold in recent years.

3.1 Insufficient testing

Many developers fail to recognize that J2ME devices can behave inconsistently. It is of utmost importance that J2ME applications are tested on many different devices. We have seen some businesses base their testing on 4–5 devices, only to be really surprised when their application does not run correctly on other devices. In our opinion, too many businesses neglect the testing face, and let their customers do the beta testing. To survive with such a strategy, you need a pretty unique application and your customers have to be both enthusiastic and understanding.

In order to get a real understanding of how J2ME phones operate, several devices from each major vendor must be tested and each version of a vendor's development platform must be represented in the set of test devices. Of course, testing 40–50 (or even more) devices will cost you. Businesses therefore need to carefully consider how much resources to spend on testing, versus the risk involved in releasing an application to customers with devices that have not been tested.

3.2 Permanent bugs

On the desktop, we're used to download patches from Microsoft Windows Update or similar systems, to solve security issues and fix bugs in our software. How is bug fixing handled on mobile phones? The short answer is that it's not. Mobile phone vendors release new software versions for their mobile phones, but these do not reach consumers that have already bought phones. The majority

of mobile phones must be handed in to a repair shop to perform a software upgrade. Only hardcore mobile phone geeks actually do this, hence the bugs you get when buying a mobile phone usually stay there for the phone's lifetime. However, if you buy the same phone a year later, you will probably get a newer software version. As mobile phone viruses have started to appear, we really see the need for a solution to the patching problem that enables consumers to upgrade the phone software themselves, similarly to what they hopefully do with their desktop computers.

3.3 Resource management

The mobile platform differs from regular desktop computers, by having limited amounts of memory, processing power, network bandwidth, and disk space available to application developers. Though smartphones have more resources than regular mobile phones, they should still be considered a resource constrained platform compared to the desktop. In addition to the traditional functionality focus, mobile developers must consider effective resource utilization to make user-friendly mobile applications.

Defensive programming is the key to create a well functioning application. For example, available runtime memory and the amount of storage memory on the device can be queried during program execution. The developer should always make sure that there is enough memory to carry out the operations of the program. If the device runs out of memory it will show an error message and terminate the application, giving the user the impression that an error occurred, while the real problem was that the application did not adapt to the amount of available resources.

3.4 Responsive applications

Applications must be responsive to provide a positive user experience. To achieve this goal, intimate knowledge of the inner workings of J2ME devices is very helpful, since different devices behave in different ways. One example is if the developer actively triggers a garbage collection to reclaim memory from unused objects. The garbage collector should then run in the background, with minor impact on the application. However, some devices will "freeze" for a few seconds while memory is collected, which is probably not what the developer wanted or expected.

Multi-threading is important when developing applications for mobile phones. A program's execution flow is event-driven, so the main thread must be idle and ready to handle events. Time consuming operations such as lengthy calculations or network communication should therefore take place in a separate thread to retain a responsive user interface. This should be familiar for developers with experience from developing graphical User Interface (UI) applications on the desktop, as the same considerations of UI- versus non-UI threads apply.

4 MIDP 2.0 security framework

MIDP 2.0 includes several mechanisms to secure an application and provide secure communication channels. This section gives an overview of these mech-

anisms, and explain why some of them have weaknesses.

Applications can be signed in order to obtain authenticity and integrity. Secure communication channels are realized by HTTPS connections, which in most cases rely on SSL (Secure Sockets Layer) or TLS (Transport Layer Security). In addition, MIDP 2.0 ensures that an application is not able to read other J2ME applications' persistent data unless it is explicitly allowed. However, since encrypted storage is not provided, hardware attacks exist to read application data. Even easier, on some devices you can install a file system explorer, locate the files used by J2ME, and use Bluetooth to send them to another device. It's clear that you cannot trust the storage system in J2ME with sensitive data.

Developers will probably have high hopes after reading the MIDP 2.0 specification, but sometimes things look better than they are. As we shall see, by studying the details of MIDP 2.0 one realizes that some critical security functionality is actually optional to implement on J2ME enabled devices, or can be based on insecure mechanisms. Also, once testing has commenced on real devices, developers will be disappointed to realize that mandatory security functionality is not implemented correctly on some mobile phones.

4.1 Application signing

Application signing based on an X.509 Public-Key Infrastructure (PKI) [4] is an optional part of the MIDP 2.0 specification and enables the device to verify the origin and integrity of J2ME application. Application signing is a good idea, but what should developers do when several mobile phones lack support for signed applications? Such a situation is highly unsatisfactory for m-commerce, or mobile governmental services since parts of your security architecture crumble if your application is installed on certain devices.

We tested two newly released Samsung smartphones during our project and both of them refused to install signed J2ME applications. The J2ME applications were signed with a code-signing certificate obtained from Verisign Inc. The Nokia devices we tested validated the certificate and applications without problems. This illustrates that to support all devices, both signed and unsigned versions of applications must be published to customers, effectively making the "signed application" feature of the security architecture optional. Since the validation of signed applications depends on a pre-installed list of root certificates (belonging to Certificate Authorities) other issues surface, which we'll discuss later.

4.2 Secure communication

Secure connections in MIDP 2.0 must be implemented by one or more of the following specifications:

- HTTP over TLS (RFC 2818) with TLS Protocol version 1.0 (RFC 2246)
- SSL version 3.0 [5]
- WTLS (Wireless Transport Layer Security) [6]
- WAP (Wireless Application Protocol) TLS Profile and Tunneling Specification [7]

Note that a developer cannot know which specifications are implemented on a specific device, without actually testing it. In the case where WTLS is used, end-to-end encryption is not provided. Secure connections will then exist between the phone and the WAP gateway, and further from the gateway to the final destination. Hence, the gateway has access to unencrypted data and must be fully trusted. This is not acceptable for a high security system, as the gateway is usually operated by the mobile network provider.

Observe also that secure connections in MIDP 2.0 require the server to have a valid certificate for authentication. However, there is no support for certificate based authentication of the client, hence the client must be authenticated on the application level by other means.

4.3 Certificate management and verification

All certificate verification procedures on a mobile phone rely on a set of pre-installed root certificates on the phone, equivalent to what we see in web browsers. Of course, different mobile phones may have different root certificates installed.

Self-signed certificates can be installed on smartphones, which is very useful during a test phase. We successfully installed a self-signed X.509v3 certificate on the Nokia 6600 by publishing it to a web server and then downloading it to the phone via WAP. A certain level of user interaction was needed after the certificate was installed, as all certificates in the Nokia 6600 have properties describing their area of use. A certificate installed by the user must therefore be enabled for verification of signed applications or server authentication before it can be put to use.

An important requirement when working with certificates is the ability to validate a certificate. Time limited validity is one mechanism, but support for certificate revocation is much more critical in order to establish a certificate's validity. The MIDP 2.0 specification states that "Certificate revocation can be performed if the appropriate mechanism is implemented on the device. Such mechanisms are not part of MIDP implementation and hence do not form a part of MIDP 2.0 security framework." Consequently, the validity of a certificate can only be established based on the assumption that none of the certificates in its certificate chain have been revoked.

One interesting observation we did was that the Samsung SGH-E720 had Verisign class 1, 2, 3 and 4 root certificates installed that were all reported valid from 1. October 1999 to 1. January 1970. Other installed certificates showed sensible validity intervals. We located the very same Verisign root certificates on the Nokia 6600, and they all had expiry date 01.10.2049. We give Samsung the benefit of doubt, and assume that the invalid expiry date is not the value actually stored in the certificate, but more of a presentation problem. However, since the date is not presented correctly, the expiry date might not be interpreted correctly during the certificate validation process. Unfortunately, we were not able to verify this, since it is not a trivial task to find a website that uses a certificate signed by a specific root certificate.

4.4 Secure storage

Encrypted storage is not supported in MIDP 2.0. An adversary may therefore use equipment to read the memory in your mobile phone and get access to

your data. Cryptographic libraries for Java exist, and may be a solution to the plaintext storage problem. One example is Bouncy Castle's lightweight cryptography API supporting several symmetric ciphers. However, the availability of cryptographic APIs does not automatically solve the plaintext storage problem. Encryption is a computational intensive task, and encryption implemented in Java can prove to be time consuming since low-level optimizations (e.g. for CPU architecture) is impossible. Native libraries or cryptographic hardware are likely to perform encryption more efficiently.

Another important issue is the lack of sources of randomness in J2ME implementations. A strong cipher can be used, but it is essential that the encryption key is impossible to guess. MIDP 2.0 does not provide a cryptographically strong Pseudo Random Number Generator (PRNG) similar to the `SecureRandom` in J2SE. Hence, the PRNG in J2ME is not suitable for generating encryption keys. Still, developers use the PRNG for different purposes which can have a major impact on the security of a system, especially since developers tend to seed the PRNG with the current time [8, Ch. 10]. One example is an attack on SSL in Sun's MIDP reference implementation [9].

4.4.1 Homemade crypto

Since MIDP 2.0 does not provide encrypted storage, and MIDP 1.0 does not provide HTTPS links, some J2ME development companies decide to specify their own "lightweight" crypto schemes. Well-meaning efforts to create new, cryptographic algorithms usually result in solutions that keep data hidden from average users, but the solutions are very seldom cryptographically strong. Such initiatives usually rely on the secrecy of the encryption algorithm, which is considered very bad practice [8, p. 268].

In many countries there are laws regulating how private data must be protected during transportation over a medium not controlled by the two communicating parties. These laws often require that the data is encrypted with an algorithm of strength equal to or better than 3DES. The Advanced Encryption Standard (AES) fulfill this requirement, but usually homemade crypto solutions will not have the strength of well tested encryption algorithms such as 3DES or AES.

5 Using clients to attack the server

We'll not consider well-known Internet server attacks from viruses and worms, or DDoS attacks [10]. Instead, we'll briefly discuss how a client application can attack the server application.

Client applications should be assumed to be evil by nature, even though you wrote them yourself. Several approaches can be used to attack a server application. Two examples are clients sending commands out of order or sending unexpectedly large data chunks as input to an application. We've talked to several businesses that develop client-server applications, and they all seem to completely trust their client software. Their arguments are along the line of: "Hey, we wrote it. Why shouldn't we trust it?" In our opinion, this is a dangerous approach as trust is easily misplaced unless a careful analysis of the trust model is carried out [8, Ch. 12].

The client software may be handed out to all kinds of people, including the ones that cannot resist trying to break it. Software can be reverse engineered, and the source code can be studied. Reverse engineering might not even be necessary, the binary code could just as well be tampered with. The gaming industry is experiencing this on a daily basis, as games are cracked to avoid license key checks. Another approach is to use a network sniffer, figure out the application protocol, and write your own malicious client that behaves similarly to the original client. Several measures can be taken to increase the level of security in an application, but it all starts with the attitude of the developers. For developers interested in building secure software we recommend [8].

6 J2ME security in the future

Several of the shortcomings in the MIDP 2.0 security architecture are addressed by SATSA, the new Security and Trust Services API for J2ME [11]. SATSA relies on a Security Element (SE), implemented in either software or hardware. This implies that the SE can take different forms such as a software component, dedicated hardware in the device, or a removable smart card. Several SEs can be available in one device. The exact form of the SE is transparent to the application developer, the interaction with the SE is handled by the SATSA implementation.

The support for cryptographic smart cards is of particular interest to developers writing J2ME applications for smartphones. Keys and certificates can be stored on the smart card, and data can be signed without the private key ever leaving the card. High-end smart cards are tamper resistant and provide authentication schemes, such as requiring a PIN or a password before access to the smart card is granted. This way, security is dependent on the smart card not being compromised. Private keys do not have to be stored on diverse insecure clients, enabling vendors to focus on keeping the smart card secure from physical tampering and, just as important, smart card API exploitation [12].

An interesting observation is that many banks are already giving their customers smart cards, which also have a magnetic strip in order to be compatible with old ATMs. By giving customers smart cards with cryptographic tools, a bank could have client software for mobile phones, PDAs, and desktop computers, all relying on the customer's smart card, hence giving (nearly) the same level of security for key storage on all platforms. Of course, different OSs have different levels of security, so a careful analysis of each platform must be carried out to make sure that the smart card is accessed in a controlled manner.

6.1 SATSA APIs

The SATSA specification defines four APIs, SATSA-APDU, SATSA-JCRMI, SATSA-PKI, and SATSA-CRYPTO. The first two APIs add functionality for smart card interaction. SATSA-APDU enables communication with smart cards using the Application Protocol Data Unit (APDU) protocol defined by the ISO7816-4 specification. SATSA-JCRMI enables high level communication with smart cards through the Java Card Remote Method Invocation Protocol (JCRMI).

SATSA-PKI enables applications to request digital signatures from an SE, hence providing authentication and possibly non-repudiation [4, pp. 32–33] by

using keys stored on a smart card. Client certificate management is also provided by SATSA-PKI, giving an application the opportunity to add or remove certificates from an SE. The most interesting part of the certificate management is the possibility to request generation of a new key-pair and then produce a Certificate Signing Request. The fact that the client generates its own keys is one of the key factors needed to support non-repudiation in a system. Note that key generation is dependent on the SE, the SE might not support key generation at all. Hence, the SE must be chosen with care, considering the application requirements.

SATSA-CRYPTO offers cryptographic tools like message digests, digital signature verification, and ciphers. The API enables applications to store data encrypted and signed on a mobile device, ensuring both confidentiality and integrity. Applications that require secure storage of highly sensitive information can therefore be realized. Note that it is up to the implementor to decide which ciphers and digest algorithms to include. The SATSA specification recommends DES, 3DES, and AES as symmetric ciphers, RSA as asymmetric cipher, and SHA-1 as the digest algorithm. SHA1withRSA is the recommended algorithm for digital signatures.

6.1.1 Security issues

SATSA is distributed as a part of the Java Runtime Environment (JRE) in some smartphones. The OS of the mobile phone must therefore be fully trusted, as the JRE depends on services from the OS. The SATSA specification states that both SATSA and the application using SATSA must trust the OS. The specification further states that when SATSA is taking over UI control, e.g. when asking the user for the smart card pin code, the UI must be "distinguishable from a UI generated by external sources" in order to prevent a malicious application from mimicking the SATSA UI. Another requirement is that "external sources are not able to retrieve or insert PIN data". These requirements put a lot of responsibility on the OS. We see three scenarios that could cause trouble.

Since the PIN is entered on the device through the keypad, a keylogger application could possibly obtain the PIN. Hence, the OS must limit the distribution of key pressed events to the SATSA implementation exclusively. Keyloggers exist for Symbian OS versions prior to version 9, so this may be an issue. Another approach would be to read the PIN from memory. When a PIN is entered, it must be stored in memory somewhere before it is handed over to the smart card. It could be possible for other applications to read this memory.

J2ME applications use a per-application dedicated logical channel to communicate with the smart card. This channel could be hijacked using low-level functionality of the OS. If access to the smart card is acquired on a level below the SATSA implementation, requests could be sent to the smart card circumventing the SATSA implementation. If this functionality was included in e.g. a Trojan horse, an attacker could have full access to your smart card without your knowledge.

We do not know if these scenarios actually apply, but they illustrate some problems with the complete trust in the OS. In our opinion, an application cannot be more secure than the OS it runs in. It remains to be seen how the mobile platform copes with viral threats, as we've only seen the beginning of viruses for mobile phones.

The Trusted Computing Group has an initiative to make OSs on mobile devices more secure (`www.trustedcomputinggroup.org/groups/mobile`). In the future, we expect that mobile devices will become more trustworthy, making it easier for developers to create secure applications.

6.1.2 SATSA shortcomings

Client certificates generated by the SATSA implementation cannot be used for authentication during setup of HTTPS links using SSL or TLS. Developers must therefore handle certificate-based authentication of clients themselves. One alternative is to open an SSL connection to a server, which authenticates the server and provides a secure communication channel. Client authentication can then be carried out using a certificate provided by the client application, and having the client sign a challenge from the server. This scheme is more robust than the widely used password authentication, as dictionary or brute-force attacks are avoided. Consequently, SATSA can improve the level of security in current server-client applications by replacing old-fashioned authentication schemes. However, it would be convenient if client certificates stored by the SATSA implementation could be used by SSL or TLS implementations found in most newer smartphones.

Signature verification with SATSA is not as easy as signature creation. SATSA generates signed messages on the Cryptographic Message Syntax (CMS) format, but does not easily validate these messages. To verify a signature, the application needs to split a message into respective data and signature parts, and must supply a public key to be used for verification. Libraries exist for handling CMS messages, so developers need not implement parsing of CMS messages themselves. However, it would be easier if SATSA could verify its own signed messages.

Signature generation and signature verification is not handled in the same way in terms of how information is presented to the user. When data is signed by the user, the underlying SATSA implementation takes control of the UI and presents the user with the certificate to be used for signing, along with the data to sign. The user can then be confident that he signs the intended data, and not something else. Signature verification is just as important, but here the application must present details about the signature to the user. This means that you trust SATSA when signing data, while you trust the application to show you correct information when verifying signatures. We would like to trust SATSA on both occasions.

Note also that SATSA does not provide verification of certificates. The developer must implement the certificate verification process and public key extraction from the certificate, along with presenting the certificate and signed data to the user. SATSA provides the most basic building blocks for PKI enabled applications, but does not include any certificate validation functionality present in J2SE.

Private keys stored on the smart card cannot be used for decryption, as they are accessible only for signing. SATSA supports asymmetric crypto, and it would be convenient if a certificate (and its corresponding private key) could be selected for decryption. Data could then be decrypted on the smart card, without exposing the private key to the application nor the operating system.

7 Conclusions

The security model in J2ME has its limitations and may not be adequate for all purposes. In addition, hurdles exist because of the bugs and limitations in various J2ME implementations on real devices. A proper analysis of the security requirements for an application must be carried out considering these limitations and difficulties.

SATSA empowers the mobile platform with cryptographic capabilities and is therefore a much needed library for secure application development. Local encryption is possible, and authentication schemes can rely on public key cryptography instead of the usual username and password authentication. Storage of user certificates and support for digital signatures in smart cards open up possibilities for applications that require a high level of security. SATSA fits perfectly in a scenario where strong authentication of the client is needed, as well as digital signatures on behalf of the client. However, SATSA provides only the basic cryptographic building blocks to build a PKI. Important functionality found in J2SE such as certificate parsing, validation, and storage is left to the developer to implement. Hence, implementing signature validation and public key cryptography based on public keys in certificates can prove to be complex tasks.

Acknowledgment

We'd like to thank World Medical Center for their cooperation during the project and for giving us access to the mobile phones needed to carry out the security related tests.

References

[1] Canalys, "Worldwide smart phone market soars in q3," last visited: March 26, 2006. [Online]. Available: http://www.canalys.com/pr/2005/r2005102.htm

[2] telecoms.com, "Mobile games industry worth usd 11.2 billion by 2010," last visited: March 26, 2006. [Online]. Available: http://www.telecoms.com/itmgcontent/tcoms/search/articles/20017303052.html

[3] JSR 118 Expert Group, *Mobile Information Device Profile for Java 2 Micro Edition, Version 2.0*, 2002.

[4] C. Adams and S. Lloyd, *Understanding PKI*, 2nd ed. Addison-Wesley, 2003.

[5] Netscape Communications, *The SSL Protocol,Version 3.0*, 1996.

[6] Open Mobile Alliance, *Wireless Transport Layer Security Specification*. WAP 1.2.1 conformance release, 2000.

[7] Open Mobile Alliance, *WAP TLS Profile and Tunneling Specification*. WAP 2.0 conformance release, 2001.

[8] J. Viega and G. McGraw, *Building Secure Software*. Addison-Wesley, 2002.

[9] K. I. F. Simonsen, V. Moen, and K. J. Hole, "Attack on sun's MIDP reference implementation of SSL," in *10th Nordic Workshop on Secure IT-systems (Nordsec 2005)*, 2005.

[10] J. Mirkovic, S. Dietrich, D. Dittrich, and P. Reiher, *Internet Denial of Service Attack and Defense Mechanisms*. Prentice Hall, 2005.

[11] Sun Microsystems, "Security And Trust Services API for J2ME," last visited: March 26, 2006. [Online]. Available: http://java.sun.com/products/satsa/

[12] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptograpic processors—a survey," University of Cambridge, Tech. Rep. 641, 2005. [Online]. Available: http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-641.pdf