Anubis: An Attestation Protocol for Distributed Context-Aware Applications

Senaka Buthpitiya, Feng-Tso Sun, Heng-Tze Cheng, Patrick Tague, Martin Griss, Anind K. Dey

Department of Electrical and Computer Engineering, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA, USA.
{senaka.buthpitiya, lucas.sun, hengtze.cheng, patrick.tague, martin.griss}@sv.cmu.edu
anind@cs.cmu.edu

***Abstract*—Sharing sensitive context information among multiple distributed components in mobile environments introduces major security concerns. The distributed sensing, processing and actuating components of these applications can be compromised and modified or impersonated to extract private and confidential information or to inject false information. In this paper we present the Anubis protocol for remote code attestation and access control of distributed components using remote execution of trusted code. Our Anubis protocol leverages previous work in the fields of wireless sensor networks and secure web-browsing. Anubis allows new components to be introduced to the environment without updating existing components. Our implementation of Anubis in Android G1 based applications shows that the protocol introduces manageable overhead (less than 600 ms latency and 35 kB packet overhead) which does not significantly impact the user experience.**

I. Introduction

Many distributed software architectures for context-aware applications, i.e.- applications that sense the context or situation of the user and adjusts their behavior proactively, have been proposed and a large number have been successfully employed in specific applications [1]. These distributed architectures are becoming even more appropriate as applications begin to take advantage of the rich array of sensors that are appearing on mobile computing devices and in smart infrastructure as mobile computing and ubiquitous computing become feasible mainstream technologies [2]. Especially when a multitude of sensors are used in conjunction with virtual sensors such as calendar services, map services and weather services [3]. Furthermore with the immense quantities of raw data gleaned from sensors, a distributed architecture is required to provide the processing power and the storage capacity needed to effectively utilize them in decision making processes. The trend to combine context information from devices and distributed computing services can already be seen with the growth of cloud computing services [4]. As components of applications are deployed on various platforms and devices some of which are in physically insecure locations these devices are susceptible to capture. Components running on a captured device are susceptible to modification, replacement by malicious components, information extraction, and/or information injection, putting the security and privacy of the entire context-aware application and its user at risk. Therefore it is essential to ensure that, 1) components are tamper proof or 2) any modifications to components are detected and compromised components are avoided.

Recent research in context-aware application security has focused on specific applications [5], but it remains an aspect of distributed context-aware architectures that has received little attention. However, distributed system security has been an area of considerable study, including wireless sensor networks [6], and distributed computing on untrusted infrastructure [7]. The techniques used in these fields become untenable when considering context-aware applications in general as very few assumptions can be made about the physical and virtual environment, especially in dynamically changing mobile computing environments where connectivity is not guaranteed. Specifically, distributed components have neither a reliable virtual connection to a trusted authority nor guaranteed physical connections to components on neighbouring devices.

In order to address the immediate threats due to component compromise, we present the Anubis protocol to protect the integrity of distributed components and ensure that context information is shared only with authorized components. The Anubis protocol is based on the principle that software components can be initially authorized offline and then attested locally between communicating components using remote execution of trusted code. The only requirements placed by Anubis on a device are that it 1)should contain enough computational power to perform asymmetric key cryptographic operations, and 2)has sufficient storage capacity to hold signatures of its own code and extra code segments used for remote code execution. Though Anubis does not explicitly require a Trusted Platform Module (TPM) [8] on each device, Anubis can take advantage of such hardware when available.

In this work, we make the following contributions toward the problem of ensuring component integrity and controlling the exposure of sensitive context information:

- We present the Anubis protocol for online attestation and access control of distributed software components in the absence of a trusted third party.
- We implement the Anubis protocol over a sample distributed software application on an Android-based mobile computing platform.
- We show that the Anubis protocol requires minimal computation, communication, and storage overheads.

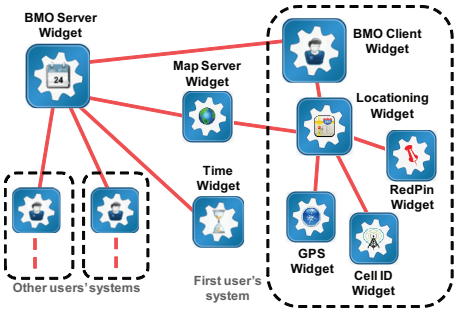

Fig. 1. Decomposition of the BMO application into widgets

II. Models and Assumptions

In this section, we present the assumptions made in the design of the Anubis protocol, the trust model Anubis creates and the types of attacks it repulses.

A. Distributed Application Model - Widget Model

While the Anubis protocol can be applied to most distributed computational applications in general, in this paper we focus on using Anubis to secure distributed context-aware applications. The context-aware applications we consider use the *widget* model presented by Dey et al. [1]. The *widget* model is widely used [9] in context-aware applications as it usefully encapsulates how distributed context information is sensed, processed and utilized in mobile environments. In the widget context model, an application is divided into components called widgets, each in charge of a particular type of context information. Widgets can run on a single device, but more common in practice is for widgets of a single application to execute on separate devices. Widgets dynamically connect with each other usually forming a tree-like topology, where the leaf nodes encompass sensors (hardware and software), the internal nodes represent filtering, inferring and/or the aggregation of information, and the root node encompasses a user interface or actuator. To exemplify this concept we present a widget based design of a Business Meeting Organizer (BMO) application that uses the context information of individuals registered with the system to intelligently arrange meetings by selecting convenient venues and times [10]. The decomposition of the overall application into individual widgets is depicted in Figure 1. The locationing widgets and BMO Client widget reside on the mobile device while all other widgets reside on servers and in the cloud.

B. System Model

In this work, we make minimal assumptions about the hardware on which widgets exist and execute, but are more concerned about the software existing on the system on which each widget resides. We assume that a device has sufficient computing power to verify public-key signatures, and that a device has sufficient memory to store a public key, a couple of hash values and code files used for remote execution. We also assume that widgets use a protocol similar to the Authenticated Diffie-Hellman Key exchange protocol [11] to set up a secure channel of communication before the authentication process begins. A further assumption is that any software in the system hosting a widget is tamper free or that the operating system hosting the system is tamper free. In applications that require a higher level of security, where this assumption is not tenable, we assume that the widgets exist on devices equipped with TPMs. We assume that widgets interested in a certain entity[1] will always be certified by authentication services that share secret credentials, which is not a unique assumption for authentication schemes used in distributed applications.

C. Trust Model

Anubis' trust model is built upon a widget securely ensuring another widget's application logic is unchanged from the point in time at which the second widget was signed. Before deployment a widget is submitted to an authentication service which verifies whether the widget should have the authority to provide or consume context regarding a particular entity, then it generates signed authentication information about the widget. This information is stored within the widget. At runtime when a widget requests a connection to another widget, widgets perform checks on each other. The widgets provide the other party signed authentication information. A widget also provides the other party the opportunity to check if the widget is in compliance with the authentication information it provided. We assume a stronger trust model where a widget performs a check on another widget to ensure it is unchanged since it was signed by an authentication service. Therefore the reliance on a third party is limited to a one time offline interaction. This is one of the major advantages of Anubis over existing work.

D. Attacker Model

Here we present attackers with two types of intent (encompassing the most common types of attacks on distributed context-aware applications), 1) to extract secret information from the system or 2) inject false information into the system. We consider computationally bounded attackers who cannot break basic cryptographic primitives. Attacker can eavesdrop, intercept, and manipulate any transmitted message. Attackers do not have physical access to the device that holds the context information which they want to extract. The attacker has physical access to any other widget in the system for a limited amount time, enabling modification or replication of those widgets. In these attacks we assume that the attacker only has limited access to widgets. In this paper we also describe features of Anubis which will increase its robustness against attackers who have continuous access to devices hosting widgets.

III. Anubis

Anubis is a protocol that allows loosely coupled components in a distributed context-aware application to remotely attest to their code and provide access control to context information.

[1]An entity is any location, object, animal or person whose context information is of some interest.

Anubis utilizes remote hashing to securely generate a signature of a component and compares it against a signature of the expected component. The expected component signature is signed by an authentication server before the component is deployed. Therefore Anubis does not rely on an omnipresent trusted third party for attestation. Our key insight is the use of remote hashing to gain a trustworthy signature of the current state of a remote component. A pair of components that successfully complete the Anubis protocol with respect to an entity, can reasonably assume each component has access rights to that entity's context information and that the components are tamper-free.

A. Logical Architecture of a Widget

The Anubis system logically divides up a widget into four separate sections (Figure 2). The logical sections are, 1) **Code Area** (contains the widget's application logic), 2) **Data Area** (holds all application variables and context information) 3) **Code Interpreter** (an interpreter with a simple instruction set, used by other widgets in the authentication process of this widget) 4) **Code Interpreter Data Area** (area for variables belonging to scripts running on the code interpreter).

B. Anubis Protocol Overview

The Anubis protocol has two stages of operation, the signing stage and the authentication stage.

Signing Stage: The signing stage of a widget occurs off-line before deployment, by an Authentication Service (AS). The AS must have the credentials to sign on behalf of the entity whose context information the widget will be handling. To perform the signing the AS will first obtain separate hash values for the code area and the data area of the widget, concatenate them and sign the string using a private key (unique to the entity whose context information will be handled by the widget). The service will also extract "code files" from the widget and sign the code files with the private key. The code files are simple scripts that are sent for execution on other widgets to authenticate them (see section on Code Interpreter and Code Files for details). Finally the AS will store the signed hash codes and signed code files along with the public key required to decrypt them within the widget's data area. See Figure 2. In applications that require greater security and execute on devices containing TPMs, the signing service captures the values of the Platform Configuration Registers (PCR) of the TPM, and combines them with the hash codes described above to create signatures of the widgets.

Authentication Stage: The authentication stage is triggered when a widget (widget A) discovers another widget (B) from which it requires context information and they setup a secure channel [11] through which A requests a connection from B. The authentication request sent by widget A contains the signatures of itself created and signed by the authentication service. In response widget B will send a code file (randomly selected from B's code file bank), which A must decrypt using the public key it received during the signing process and execute the file on its code interpreter. The code file creates hash codes of the code and code interpreter areas of the widget, and of itself (when TPMs are employed the code file captures the values of the PCRs and combines them with the hash codes). The code file will proceed to check the return address of the code interpreter at the end of executing the code file (i.e. - the first instruction within the widget scheduled to be executed after the code file finishes execution), to ensure it points to an address within the code area of widget A. Checking the return address of the interpreter to be within the code segment being hashed, ensures that the code file is not hashing a "dummy copy" of an untampered widget kept in the data section of a modified widget. Finally the code file will proceed to place a timestamp and the three hash codes it generated in a packet with a format unique to that code file and transmit the packet to widget B. The timestamps prevent compromised widgets from using transmissions from earlier authentications (before being compromised) to forge authenticity. From the moment that widget B transmits the code file to widget A, B will begin a countdown to effect a time-out after a predefined interval, thus ending the attestation process in failure. The predefined interval is decided by the designer of widget B, and it is recommended to keep the interval as low as possible while allowing time for the back and forth transmissions and execution time of the code file. Once B has received the packet from the code file, it will extract the three hash codes and ensure that the hash code of the code file has the expected value. Next Widget B will decrypt the initial string received with the authentication request, which contains the hash codes of the code area and the code interpreter calculated by the authentication server. Since widget B also handles context information of the same entity that widget A is interested in, widget B should also be signed by an AS with credentials (private and public keys) of that entity. Therefore B would have received the public key required for verification of the string sent by A along with the attestation request. Widget B will extract the hash codes from the decrypted string and compare them against the hash codes calculated by the code file. If the hash codes match, then B can safely assume that A is tamper-free and authorized to access the entity's context information. The attestation process of widget A is illustrated in Figure 3. Once B has authenticated A, widget A will proceed to authenticate B using the same process in reverse. The authentication will time-out after a random period of time (triggered by either of the two widgets), and after a time-out the widgets must re-authenticate using the same procedure (the maximum interval for authentication validity is a trade-off between communication/time cost of authentication versus the criticality of information held by the widget).

C. Code Interpreter and Code Files

The code interpreter is a simplistic interpreter used to execute code files received from other widgets during authentication. Code files running on the interpreter have three tasks, i.e. - to calculate hash codes of sections of the widget, to check the return address of the code interpreter to the main body of code, and transmit the calculated values to the code file's

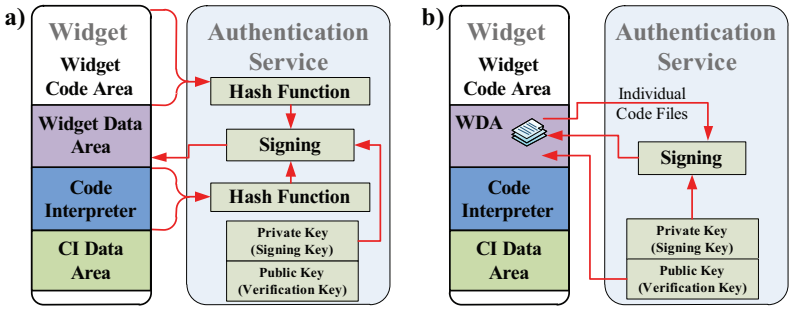

Fig. 2. Signing process of a widget

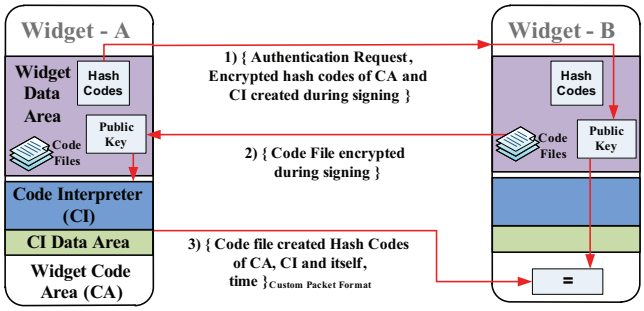

Fig. 3. Authentication process of widget A by widget B over a established secure channel

parent widget. The code interpreter must be flexible enough to allow the code files to perform their tasks, while protecting the interpreter's widget and data. To this end we specify a set of rules that an interpreter must enforce.

1) Disallow writes to any part of the widget's memory except to the interpreter's data area, thus protecting the widget's data and code.
2) Block access to the widget's data area to prevent the code file from pilfering information.
3) Read-only access to the widget's call stack, allowing checks of the return address of the interpreter.

Methods to enforce these rules are very dependent on the device hardware, firmware and operating system on which the widget exists.

Each widget has multiple code files from which it chooses one at random to use when authenticating another widget. As the number of code files carried by a widget increase, the Anubis protocol's authentication strength against a widget that is continuously controlled by an adversary also increases.

D. Authenticating for Multiple Entities

In cases where a widget is interested in context information belonging to more than one entity, Anubis can simply be implemented separately for each entity. The widget would be separately signed by authentication services that have credentials for each entity. When two widgets interested in multiple entities authenticate each other, they repeat the authentication process for each of entities that they want to share context information about.

IV. Implementation

We have implemented a prototype version of the Anubis protocol and created a sample application which uses the protocol, as proof of concept and for collecting performance statistics. The implementations are on commodity laptops running Ubuntu Linux and HTC Android G1 phones. The Android platform was selected as a proof of concept for operation in mobile computing environments.

A. Prototype Implementation

The prototype implementation consists of three separate components, 1) the data source/server widget, 2) the data consumer/client widget, and an authentication service component. The prototype is implemented in C++ and compiled using the standard GCC compiler collection in Linux (Android based components were compiled using a GCC compiler with a custom build environment).

Authentication Service: The Authentication Service (AS) is an application designed to execute in the widget's native platform once, before the deployment of the widget. It also has the ability to generate credentials for an entity, in the form of a pair of public and private keys (for signing and attestation) using an Elliptic Curve Integrated Encryption Scheme (ECIES) [12]. Our implementation of the AS acts in the same manner as a separate widget would to authenticate a widget. First the authentication server executes the widget and then invokes an authentication, during which the widget is sent a code file to create hash codes (using the SHA-1 hash algorithm) of the internal structure of the widget. Then AS kills the widget's process and signs the hash codes using the entity's private key with the ECIES scheme. Next the widget's executable file

is opened by the AS to insert the signatures and the entity's public key into empty space (identified by unique place holder values) assigned during widget development. Then the code files are extracted (located using predefined marker values), signed and replaced. In out prototype we use three unique code files per widget.

Widgets: The implementation of the Widgets do not change drastically when the Anubis protocol is implemented, apart from allocating extra memory locations (filled with place-holder values) to hold signatures and keys provided by the authentication service. The widgets will also hold a code interpreter, which is common across all widgets.

Code Interpreter: Our interpreter implementations simulate the platform on which the widget itself is executing. In a previous section we discussed the rules an interpreter must enforce to ensure a code file does not act maliciously, in our implementation we check that the code files adhere to these rules during the signing process in the AS.

Cryptographic Algorithms: The hash function used in our implementations is SHA-1 which provides a 160 bit hash code. The public key cryptography system used in this implementation is an Elliptic Curve Integrated Encryption Scheme (ECIES). The decision to use these algorithms (as opposed to RSA and SHA-2) are based on the trade-off between encryption key size and computational overhead versus the security gained.

B. Business Meeting Organizer Implementation

The Business Meeting Organizer (BMO) system, described earlier was implemented to use just the location information of the users when arranging a meeting. Once the BMO widgets were implemented each set of widgets belonging to just one user were signed with his credentials. The central decision making widget (BMO server widget) was signed repeatedly with each users' credentials. In order to introduce a new user into the system, the decision making widget has to be re-signed with the new user's credentials.

V. Evaluation

A. Method

In the first performance evaluation we use the Anubis system on two widgets executing on two Dell Latitude D630 laptops running Ubuntu Linux, while in the second evaluation we use two Android G1 mobile phones. We characterize processing overhead and data exchanged during authentication against the size of the widget being authenticated.

We keep the size of one widget (A) constant while the size of the second widget (B) is changed. For each size of B, a set of 10 readings for authentication time and communication overhead are measured and averaged. The size of B was increased by inserting code to perform place-holder calculations, and this additional code is not called before or during the authentication process. This performance data is gathered over an unencrypted network to capture the true overhead of the protocol. To capture the true overhead of the protocol, data is gathered over an unencrypted network.

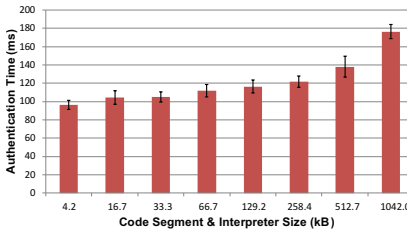

Fig. 4. Variation of authentication time of a widget vs. widget size when execution is on a laptop

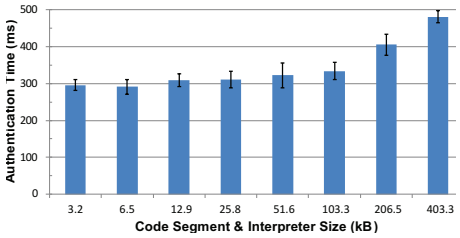

Fig. 5. Variation of authentication time of a widget vs. widget size when execution is on an Android G1 phone

B. Results

As can be seen from Figures 4 and 5, the authentication processing time increases as the size of a widget increases. This comes as no surprise as the amount of binary code the code file has to hash increases with the size of the widget. The code segments and code interpreter sizes shown in Figures 4 and 5 are in words, i.e. - 4 bytes in 32 bit architectures of the laptops and the G1 phones. And from our sample application we have observed that the average code segment of a widget remains within 20 kilobytes (or approximately 5000 words), which again is an estimation that overcompensates. Therefore it is safe to assume that a widget in general would authenticate in under 110 ms on a PC and under 300 ms smart-phone.

It was also observed that the data exchanged during authentication process of a single widget remained at a constant level of just below 17 kilobytes. Therefore the bidirectional authentication would have a communication overhead of under 35 kilobytes. This observation is in line with the theoretical predictions as the data exchanged are authentication requests, hash codes, timestamps, etc. which are completely independent (in size) of the size of the widget being authenticated.

VI. Related Work

The concept of component-based context-aware applications are not novel [1], but it is only with the advances in mobile computing and telecommunication bandwidth that components or widgets are being placed on physically distant devices. When components execute in a secure environment the need to authenticate is minimal, therefore only recently has this come to the forefront. Prior work in the area of security in context-aware applications[13], has largely focused on a very small application classes or on a single application, and related work with generic security in this area is limited.

There are other fields, such as wireless sensor networks, where operating environments pose similar security problems while posing slightly different constraints on security protocols. Causes for security threats faced by wireless sensor networks, described by [6], are very similar to the causes of security issues faced by distributed context-aware applications. The two main approaches in this field to deal with security issues are 1) using trusted entities for authentication [14] and 2) using of probabilistic key management [15]. The approaches focus on authenticating sensors to ensure that they belong to the network but focus only on limiting the damage caused by the compromise of a sensor node rather than attempting to detect compromised nodes. Probabilistic key management protocols such as [15], require a large number of nodes to be present with at least a minimum level of node density, which cannot be predetermined in a context-aware application environment. Perrig et al.[14] assigns certain nodes as base-stations to act as mediators for communicating nodes to authenticate each other. In our work, we deal with context-aware applications in general, and as such we cannot guarantee that a widget would have access to a trusted widget or service for authentication. Furthermore, we envision newer context-aware applications will leverage already deployed widgets, requiring new widgets to be deployed into existing networks, but the schemes proposed in [14], [15] are unable to handle this requirement.

Using hash codes for authenticating software is not a novel concept. The use of hash codes to verify software executing on remote platforms has been performed in previous work [7]. By requiring a reboot and respectively checking the BIOS, the permanent storage medium, the operating system, and finally each software component it guarantees an high level of security at a heavy computational expense. Device reboots causing widgets to, 1) loose existing connections and 2) rely on slow permanent storage to retain gathered context information, are not acceptable for our application domain. Garris et al. [7] rely on the TPM that resides on the remote device, to create hash codes, while the Anubis protocol is able can operate without a TPM being present.

Verifying the execution of code files. The techniques reviewed in [16] provides valuable theoretical insights on checking the results of a process. But are many practical limitations when implementing a real system. An interesting mechanism for ensuring a piece of code was executed securely on an untrusted host is introduced in [17]. The main limitations of this work is its high computational overhead and that it requires access to trusted services for key distribution and management, interpreter certification.

VII. Conclusion

In this work, we presented the Anubis protocol for online attestation and access control of distributed software components for context-aware applications, but it can be easily expanded to general distributed component-based software. Anubis achieves online component attestation using component signatures created at design-time and compared against actual components at run-time by a component on a communicating device. The primary benefit of Anubis is that it does not rely on the run-time presence of a trusted authority. Moreover, the system allows new components to be introduced into the distributed environment without interaction or modification of existing components. We believe that the security provided by the Anubis protocol is a reasonable trade for the marginal increase in execution time and required storage space on each component, both well within the capabilities of existing commercial devices.

Acknowledgment

This research was supported in part by a grant from the Ericsson Research Center and by the CyLab Mobility Research Center at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office.

References

[1] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications," *Human-Computer Interaction*, vol. 16, no. 2, pp. 97–166, 2001.

[2] A. K. Dey, D. Salber, and G. D. Abowd, "A context-based infrastructure for smart environments," in *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments*, 1999, pp. 114–128.

[3] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[4] J. H. Christensen, "Using restful web-services and cloud computing to create next generation mobile applications," in *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 627–634.

[5] Y. J. Song, W. Tobagus, D. Y. Leong, B. Johanson, and A. Fox, "isecurity: A security framework for interactive workspaces," Palo Alto, CA, USA, Tech. Rep., 2003.

[6] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[7] S. Garriss, R. Caceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang, "Trustworthy and personalized computing on public kiosks," in *MobiSys '08: Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services*, 2008, pp. 199–210.

[8] "Trusted computing group," http://www.trustedcomputinggroup.org.

[9] A. K. Dey, D. Salber, G. D. Abowd, and M. Futakawa, "The conference assistant: Combining context-awareness with wearable computing," *Wearable Computers, IEEE International Symposium*, pp. 21–27, 1999.

[10] K. Yang, N. Pattan, A. Rivera, and M. Griss, "Multi-agent meeting scheduling using mobile context," in *Proceedings of the First Annual International Conference on Mobile Computing, Applications, and Services (MobiCASE)*, 2009.

[11] S. Blake-Wilson and A. Menezes, "Authenticated Diffie-Hellman key agreement protocols," in *SAC '98: Proceedings of the Selected Areas in Cryptography*, 1999, pp. 339–361.

[12] A. J. Menezes and S. A. Vanstone, "Elliptic curve cryptosystems and their implementation," *Journal of Cryptology*, vol. 6, no. 4, pp. 209–224, 1993.

[13] J. E. Bardram, "Applications of context-aware computing in hospital work: examples and design principles," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 1574–1579.

[14] A. Perrig, R. Szewczyk, J. Tygar, and V. WenPerrig, "Spins: Security protocols for sensor networks," *Wireless Networks*, vol. 8, no. 5, pp. 521–534, 2004.

[15] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 41–47.

[16] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *J. ACM*, vol. 44, no. 6, pp. 826–849, 1997.

[17] G. Vigna, "Cryptographic traces for mobile agents," in *Mobile Agents and Security*, 1998, pp. 137–153.